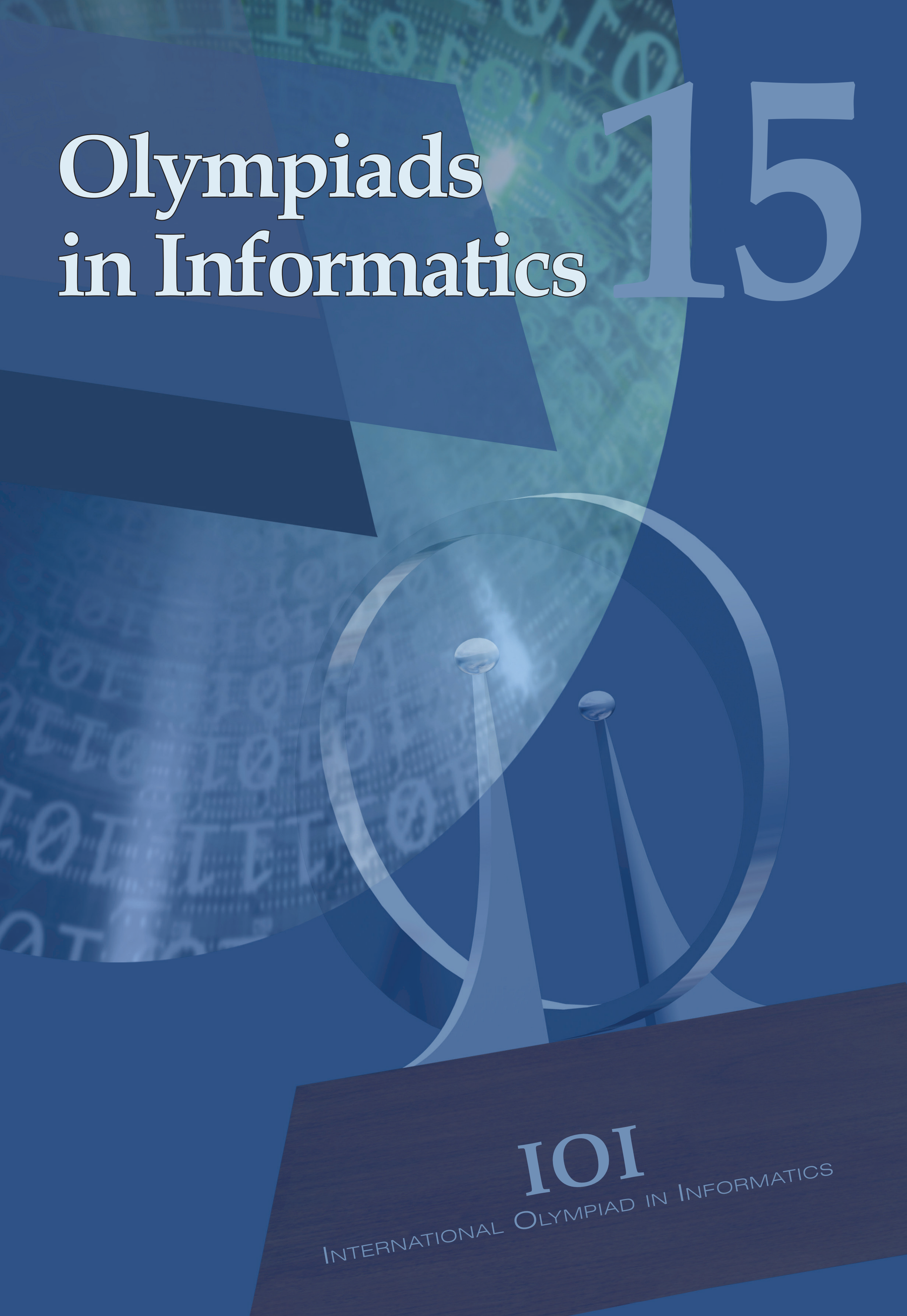
Olympiads in Informatics

15



ISSN 1822-7732

INTERNATIONAL OLYMPIAD IN INFORMATICS
VILNIUS UNIVERSITY

OLYMPIADS IN INFORMATICS

Volume 15 2021

Selected papers of
the International Conference joint with
the XXXIII International Olympiad in Informatics
(online) Singapore, 19–25 June, 2021

OLYMPIADS IN INFORMATICS

The journal Olympiads in Informatics is an international open access journal devoted to publishing original research of the highest quality in all aspects of learning and teaching informatics through olympiads and other competitions.

`https://ioinformatics.org/page/ioi-journal`

ISSN 1822-7732 (Print)
2335-8955 (Online)

Olympiads in Informatics, 2021 Vol. 15, 1–2
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.00

Foreword

IOI, the International Olympiad in Informatics, second year in a row organized by Singapore from June 19th to 28th, 2021, is held online again due to the worldwide spread of COVID-19. A traditional one-day scientific conference is replaced by only virtual presentations of the articles on June 21st.

The 15th volume consists of ten research articles, and two reports.

Giorgio Audrito, William Di Luigi, Luigi Laura, Edoardo Morassutto, and Dario Ostuni share the experience gained and tools produced during a year of online Olympiads in Italy, hoping that other countries can benefit from these tools and suggestions for their own Olympiads. The authors provide a list of online resources after the reference list, including GitHub urls of the developed tools.

In his article "Self-Generated Figures in Sequence Processing", David Ginat deals with self-generated figures in algorithmic problem solving. Such figures elicit associations of hidden patterns, whose recognition yields elegant and efficient algorithmic solutions. IOI students have demonstrated constructive utilization of self-generated figures in solving challenging sequence processing tasks. The author believes that this problem-solving heuristic should be elaborated, exemplified, and studied in the teaching of algorithmics, at all levels, including the Olympiad level.

Jamaladdin Hasanov, Habil Gadirli, and Aydin Bagiyev provide statistical analyses based on the last on-site IOI and share insights on each of them. The sources of the article and results of the work can be found in a public GitHub repository (`https://github.com/ADA-SITE-JML/ioi-grant`)

Martin Mare presents "Security of Grading Systems" and discusses various attacks on grading system security. Several recommendations are summarized in the conclusion part.

In their article, "You Should Know and Not Only Use Sorting Algorithms: Some Beautiful Problems", Laszlo Nikhazy, Aron Noszaly, and Bence Deak present some beautiful tasks where the key to the solution lies in knowing a particular sorting algorithm. In some cases, the sorting algorithms are applied as a surprisingly nice idea, for example, in an interactive task or a geometry question.

The article of Pavel S. Pankov, Taalaibek M. Imanaliev, and Azret A. Kenzhaliev deals with methods of generating various Olympiad tasks by using evident images of virtual automatic makers. Such tasks are well-understood, have short formulations and are difficult for solving even with initial data of small volume. The authors hope that these tasks would enlarge the scope of tasks involved in the IOI and give ideas for young people to implement in hardware.

The article of Zsuzsa Pluhar presents the newest extending activity idea, a challenge game of the Hungarian Bebras initiative. The goal of extension is to create unplugged computational thinking activities based on the Hungarian Bebras competition.

Vesna Dimitrievska Ristovska, Emil Stankov, and Petar Sekuloski present a comparative analysis of the conduction of traditional courses, as opposed to the conditions with distance education. The analysis is done from the aspect of the approach to teaching as well as from the aspect of exam conduction and achieved exam results. Presented scenarios for conducting online exams may also be used for conducting online contests in informatics.

Marina S. Tsvetkova and Vladimir M. Kiryukhin discuss the concept of algorithmic thinking in the context of the history of the formation of school informatics, in the competencies of new digital literacy and in the system of developmental education.

Tom Verhoeff presents an excellent article, "Look Ma, Backtracking without Recursion", and demonstrates how backtracking can be discovered naturally without using a recursive function nor using a loop with an explicit stack.

In the second part of the volume, a national report from Cuba is presented by Francisco Hernandez Gonzalez, José Daniel Rodriguez Morales, and Dovier Antonio Ripoll Mendez. The Cuban Olympiad in Informatics has different stages ranging from the school level to the national contest. In recent years, the competition has been renewed with the use of an instance of the Don Mills Online Judge, an open-source online judge.

Finally, Antti Laaksonen overviews two recently published competitive programming books: "Algorithmic Thinking" by Daniel Zingaro, and "Competitive Programming in Python" by Christoph Dürr and Jill-Jênn Vie.

Many thanks to all of those who have assisted with the volume, authors and reviewers, as well as the Editorial Board of this journal. A lot of work to be done in the process after submitting the first version of papers until the final version ready for printing. We would like to thank the host of this year's IOI in Singapore for organising the IOI conference online.

Editors

Olympiads in Informatics, 2021 Vol. 15, 3–12
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.01

The Italian Job: Moving (Massively) Online a National Olympiad

Giorgio AUDRITO[1], William DI LUIGI[2], Luigi LAURA[2,3], Edoardo MORASSUTTO[2], Dario OSTUNI[4]

[1]*Department of Computer Science, University of Torino, Italy*
[2]*Associazione Italiana per l'Informatica ed il Calcolo Automatico*
[3]*Uninettuno University, Rome, Italy*
[4]*Department of Computer Science, University of Verona, Italy*
e-mail: giorgio.audrito@unito.it, williamdiluigi@gmail.com, edoardo.morassutto@gmail.com, luigi.laura@uninettunouniversity.net, dario.ostuni@univr.it

Abstract. The COVID-19 pandemic is having a pervasive effect worldwide, including local, national and international Olympiads in Informatics. Most national Olympiads had to be moved online, a process which poses a number of serious challenges. Help across countries is of uttermost importance in this context, to enable a successful continuation of the IOI during globally hard times. In this paper, we share the experience gained and tools produced during a year of online Olympiads in Italy, hoping that other countries can take profit of these (freely available) tools and suggestions for their own Olympiads.

Keywords: team work, programming contest, Olympiads in Informatics, peer education, programming training.

1. Introduction

The *Olimpiadi Italiane di Informatica (OII)*, or Italian Olympiads in Informatics (Casadei *et al.*, 2007), are laid out in phases spanning a two-year length: the **Phase-1** of the N-th edition takes place in November–December of the year $N-1$, the **Phase-2** takes place in April of the year N, and finally the **Phase-3** takes place in September of the year N. The highest ranked students in this phase are then selected for participation in training camps during the whole scholastic year (**Phase-4**), ultimately leading to the selection of the Italian team at the IOI for year $N+1$. The selection process starts with a scholastic pen-and-paper test involving between 10k and 15k participants in Phase-1, which are then reduced to about 1k-2k participants for a regional programming competition in Phase-2. About one hundred of those students are selected for the national programming contest in Phase-3, and 20-30 of them are allowed to enter the training camps in Phase-4.

As a result of the COVID-19 pandemic, all four phases of the OII were affected and thus had to be moved to a fully-online setting. The Phase-1 of the OII 2020 (which happened at the end of November 2019) was the last onsite competition. After that, the Phase-4 of the 2019 edition, the Phase-2, Phase-3 and Phase-4 of the 2020 edition, and the Phase-1 and Phase-2 of the 2021 edition had all to be moved online, after being postponed by a few months in the hope that schools would reopen later in the year. This forced shifting required the development of new tools and solutions, which ensured an overall successful year of Olympiads in Italy. In this paper, we share these (freely available) tools and solution, for the benefit of other countries which may need it for their own upcoming Olympiads. The next three sections present our experience with the first three phases of the OII, in order. Phase-4 has not been included, since for the limited number of participants, it did not require the development of innovative solutions.

2. Phase-1 (Scholastic)

In this phase, students have to answer a "quiz-based" exam, with multiple-choice questions and (numeric) open questions of a logical or algorithmic nature. This contest is normally held in about 500 high-schools, each of them with a designated "contact person" in charge of: distributing the paper-based exam to the students, proctoring them during the exam, collecting the completed exams when the time is up, and finally computing the score of each student. The large number of participants and the "standardized-test" style of the contest made this phase a significant challenge in moving it to an online setting.

Going online. We evaluated several possible solutions to hold this phase online. We initially wanted to restrict as much as possible any possibility of cheating, so we considered using some kind of online platform that would present the questions in a random order and with a fixed time to answer each question (e.g. five minutes) while preventing to "go back" so as to discourage students from sharing answers, since they would have limited time to answer them.

In the end, we concluded that implementing this idea was going to be a significant challenge considering the available time and workforce, and the high requirements in terms of the overall load on the system. We also decided that we didn't want to disrupt the experience so much for the students by putting pressure on them to answer questions quickly and by making them unable to change their answers.

The strategy we finally went with was to stick as much as possible to the "paper-like" feel of the exam, by providing them with a simple PDF file, so the students could immediately see every question and choose in which order to solve them. However, we not only randomized the order of the questions in the file, but we also put randomized data in the actual exercises. The idea behind this was to trick cheating students into suggesting each other wrong answers (since the questions would look almost the same, but would have slightly different data) or at least spend a significant time decoding which "version" of a question they had.

Finally, to ensure that we didn't have downtime caused by the load of tens of thousands of users making requests to our servers at the same time, we offloaded everything we could to external services, like **Twitter** and **Google Forms**.

Randomization of the questions. In order to prepare questions with fully or partially randomized data, we developed a small program that we called **randomTeX** [6]. This program uses Jinja2 as the template engine (with a few configuration tweaks that were necessary to resolve some language ambiguities that came up because LaTeX and Jinja2 have some common syntax, like the opening and closing curly brackets). To feed the data to the template, randomTeX uses an approach similar to the YAML front-matter in Markdown documents.[1]

We prepared a separate LaTeX file for each of the 20 exercises in the exam, and the variations of each exercise were defined by the data in the YAML front-matter of the file itself. This approach turned out to be very flexible, since we could easily add new versions of the same exercise by changing only the exercise file very slightly and without having to touch the LaTeX code.

After some tests, we settled on generating four variations of each exercise: although we could easily have much more than four, we deemed this number enough for our needs and it ultimately made it possible for us to manually verify that each and every variation made sense. We also ended up avoiding multiple-choice questions in favour of numeric open questions, to simplify both the correction process and to minimise the information available to cheating students.

This, combined with a simple randomization in the presentation order of the questions, made it possible to generate *a different PDF for every student*. In the first page of the PDF file we provided information such as the "exam ID" and the First and Last name of the student, to make it clear that the exams were different.

Distribution of the exams. In order to facilitate distribution of the exams (while avoiding a high load on our servers) we decided to simply start distributing the PDF files well ahead of time: we provided a download link 6 hours in advance. Each PDF file, although with different content, was protected by the same password. To disclose the password to every student at the exact starting time of the contest, we used the "scheduled tweet" functionality on Twitter.

Specifically: each student downloaded their custom PDF file ahead of time and, as the start of the exam approached, he or she simply visited the Twitter account of the OII and started monitoring the page for updates. As soon as the contest started, the account automatically tweeted the password that unlocked all the different PDFs.

Collection of answers. We offloaded the problem of collecting answers to an external service: Google Forms. We created a form that was purposefully very simple: a field for the "exam ID" and 20 questions numbered from 1 to 20 without any question detail (as the order was anyway different across students) and with an open numeric field to speci-

[1] For further details, see `https://jekyllrb.com/docs/front-matter`

fy the answer. The idea was to collect the answers in the most reliable possible way, and then perform the actual validation once the contest ended. One detail worth mentioning is that we generated the "exam ID" in a way that it would be easy to reconstruct in case the student would mistype it: we simply concatenated five random words.

To reduce the possibility of last-minute connection issues that students might have had, we specified in the instructions in the first page of the PDFs that it was recommended to submit frequently: this was possible because we enabled "Edit after submit" in the form's settings.

The plan worked well for most of the contest. Unfortunately, several students didn't read or simply didn't follow our advice, and submitted right at the end of the contest: this probably triggered some malicious activity alert (the form started to ask students to solve a CAPTCHA before submitting) and this caused some students to not be able to submit their answers in time. To address this, we had to re-open the submission window (we actually duplicated the Google Form and linked it from the previous form) specifically for "late submissions" and we informed students to use the new link if they weren't able to submit their answers in time, while also reminding them that we reserved the right to accept or discard those late submissions.

Evaluating the submissions was straightforward: when we generated the PDF files we stored the list of correct answers for each file. After the contest we downloaded the spreadsheet with the answers and using the "exam ID" we compared, for each student, their answers with the correct ones. For each exercise we graded only the last submission that included that exercise (since there were no penalties for wrong answers).

Possible improvements. In case we will have to go online again next year, there are a few things that we could improve:

- Instead of using a single Google Form for all the participants, it might be better to use multiple Google Form instances (e.g., ten different forms). We can easily modify the PDF template of the exam so that each student will see in his/her exam PDF a differ*ent link* to the submission form. This might reduce the likelihood of triggering the CAPTCHA requests.
- Upon performing a statistical analysis of correct vs wrong answers for each variation of each exercise, we noticed that in one of the four variation of one specific exercise there was a significantly higher rate of correct answers. This turned out to be caused by the fact that, when we slightly changed the numbers, we introduced an "easier optimal strategy" for the exercise resolution, which was suboptimal in the other variations. Although this didn't greatly affect the results, we were very concerned by this and we are considering introducing some kind of automated validation step in the randomTeX program (e.g. by writing a validation script for each exercise which takes the YAML front-matter as input and outputs a Boolean value) which could help us verify that some conditions are true for all variants of an exercise (e.g. the optimal solution being unique, and so on).

3. Phase-2 (Regional)

In this phase, students have to complete a programming contest, grouped in *territorial* zones based on the geographical position of their school. Italy is a country divided in 20 regions, with very different populations and therefore number of schools. Since this contest was onsite, this heterogeneity used to be an issue: participants from a same region ranged from hundreds to below a dozen (e.g., from 7 to 210 participants in the last onsite contest). In order to mitigate this problem, regions had to be split into *venues* of roughly the same size (from 7 to 51 in the last onsite contest), according to the capacity of the school hosting the contest.

Unlike Phase-1, which mostly evaluates logical and code-reading skills in order to reach as many students as possible, Phase-2 is focused on selecting students that can write code to solve actual programming tasks, both theoretically and practically by writing a program that given an input file produces the correct output. Time and space complexity is not an explicit focus of this phase, as opposed to IOI-like competitions, so every problem is output-only and no explicit time limits are given.

In order to evaluate effectively these skills, the Phase-2 does *not* use the same judge system used in the IOI, but rather one that we specifically designed for this back in 2017, called **Terry** [9]. After accessing the Terry interface, the participants can choose a task and download an input file (which is unique for each student, and changes every time a submission is attempted), run their program locally with the downloaded file as input, and finally upload the produced output file along with the source code of the program producing it (for plagiarism avoidance) and immediately receive a score. This approach was heavily inspired from the early format of the Google Code Jam competition, and allows us to grade submissions of a very large number of students with relatively few resources (as we don’t need to run the students’ code, except in selected cases) and allows the students to use a broad set of supported languages.

After the contest ends, we perform plagiarism checks. We used to do them using JPlag (Prechelt and Phlippsen, 2000), which however only supports a subset of the allowed languages. For this reason we recently developed Starplag [7], that computes the similarity between two source files regardless of their programming language, by implementing a variation of the Levenshtein distance where text substitutions (i.e., variable renaming) is considered.

Going oline. This was the first phase to be moved online, since it was scheduled to take place in April 2020, right after the March COVID-19 restrictions took place in Italy. Terry was not meant to work as an online judge: instead, it was designed to be deployed “offline” in each of the onsite venues. Since every venue had a designated reference person, we could easily communicate with the students through this person: if a student had a question on the tasks, the reference person (e.g. a teacher in the venue’s school) would forward it to us, and we would then provide an answer which would be finally relayed to the student.

Since this process does not work for an online contest, we had to write a new component for Terry handling questions, answers and announcements. This new component worked well enough, although leaving margins for future improvements. In particular, it showed all the questions in a “stream” as they arrived to us, making it difficult to reconstruct the context of a question. We underestimated the importance of a “conversation” with a student: a question from a user would often implicitly reference a previous question that the user already sent us, and sometimes a question just couldn’t simply be replied with an answer and we had to “answer” the question with another question. For the next year, we plan to develop a conversation-like interface addressing this issue.

Proctoring in real-time more than a thousand students from home with available resources was unfeasible, so we instead adopted the standard approach of online contests, where only after-contest checks are performed. After the contest was over and before extracting the ranking, we examined the submitted source files looking for evidence of plagiarism or irregularities.

Technical aspects. The contest server was designed to run inside a virtual machine sent to the venue hosts, in order to require minimal technical knowledge from them. The virtual machine only had to serve that venue (around 50 students) and be as light as possible since we had no control over the quality of those servers. For this reason, Terry uses SQLite as DBMS: it is light and fast enough for our needs. Scaling a system designed to handle few dozen users to support few thousands users can be risky. To limit this risk, we decided to keep each Italian region in a separate shard of the system, with its own independent instance of Terry. This way each instance only had to process a fixed fraction of the users, also allowing us to migrate each region independently in case of failure of some instance.

We used Docker containers for shipping the replicas and docker-compose for orchestrating them. Ideally, each instance should have used its own host for full separation of the load; however, we ended up renting 8 VMs on Google Cloud Platform[2], assigning each of the 20 regions to a specific VM in a way designed to distribute the load uniformly. Besides the 8 VMs, we also set up a *coordinator* VM to host the communication component and some utility scripts. Among these extra tools, we also had an instance of Grafana[3] displaying many metrics collected by Prometheus[4]: at [4] and [3] you can find a snapshot of our dashboards. From those dashboards you can see the machines we had (named with Greek letters), where *phi* is the coordinator. The communication platform was able to handle an average of 60 requests per second, while the other VMs had an utilization close to zero for most of the contest duration. Unfortunately, due to a misconfiguration of nginx (a wrong request rate limit) the expected spike at the beginning of the contest caused nginx to terminate connections, not allowing them to reach Terry’s backend (see the “503” tab in the dashboard [4]). This was fixed in about half an hour; for the future, it will be important to stress test the system also with reasonable loads, not only with *denial of service* rate of requests.

[2] `https://cloud.google.com`

[3] `https://grafana.com`

[4] `https://prometheus.io`

Since the users were partitioned into different servers, we had to make sure that each user connected and logged in the correct container. This was accomplished through a DNS pointing each user to the correct VM, then to the correct instance of Terry within that VM. More precisely, we had separate URLs for each region (e.g., `lom.territoriali.olinfo.it` for Lombardy), adding a `CNAME` record to this domain pointing to one of the VMs, for example `alpha.territoriali.olinfo.it`, that in turn is an `A` record pointing to the VM's public IP. This extra step allowed us to easily move one container into a different VM by simply spawning it and changing the content of the `CNAME` record (a low TTL was set to support this). Fortunately, this measure wasn't necessary as the system run smoothly after the rough start.

4. Phase-3 (National)

Since 2011, the Italian national phase has used the Contest Management System (CMS) (Maggiolo and Mascellani, 2012) as its platform to host the contest; CMS is also the core of the training platform available for students (di Luigi *et al.*, 2016) and has been used also in team Olympiads (Amaroli *et al.*, 2018). The contest is IOI-like: 5 hours and (usually) 3 problems to solve. The only major differences lie in the difficulty level of the problems, which are usually easier than IOI ones (di Luigi *et al.*, 2018), and the set of allowed programming languages: in the Italian national phase only C and C++ are allowed. Before the pandemic, this phase used to be a 3-days onsite event, hosted by a different high school or educational institution each year. The about 100 students admitted to this phase would travel to the contest site with their regional contact person. After the contest, the top 20-*ish* students would continue their journey to the training camps, held in the Italian city of *Volterra*, in *Tuscany*, during which the Italian team for the IOI would be selected.

Going online. As the contest moved entirely online, we had to deal with several problems, the major one being proctoring. While during the previous phases the problem of potential cheating was relatively irrelevant and could be dealt with offline tools, with only about 100 contestants the potential problem of cheating had to be dealt with properly. Thus, much of the organization of the online contest revolved around this point.
The competition itself was still hosted with CMS, as its use onsite or online is not significantly different. We let the students use their own machines to compete in the contest and set up a 3-layered proctoring system:

1. All contestants where assigned to one of 14 different Zoom[5] meetings where they would have to be in for the whole duration of the contest, with the camera on and the microphone unmuted. A human proctor from the staff were assigned to each of them, having to look over approximately 8 contestants, while also helping as a communication facilitator.

[5] `https://zoom.us`

2. All contestants were required to compete from inside a virtual machine provided by the organization, having it full-screen for the whole duration of the contest. The virtual machine, aside from blocking access to internet, also recorded the contestant's screen and sent the resulting stream to the server.
3. All contestants were required to run a custom-made proctoring program on the host machine, called *oii-proctor* [5], which would check for misbehaviors (such as opening a browser or leaving the virtual machine) and report them.

Furthermore, in order to mitigate possible unexpected cheating behaviours, the difficulty and novelty of the problems was increased relative to previous years, and the number of people selected for the training camps was also increased to 29. This was also made possible by the simpler logistic and lower budget required by an online training camp, which used to be limiting factors for this number.

The handling of the contest was overall successful, with only some secondary issues:

- Contestants with bad internet connections or very slow PCs had, inevitably, a disadvantage.
- The increased difficulty of the problems almost halved the average score: in 2018 it was 79, in 2019 it was 82.34 and in 2020 only 49.7; this also caused the cut for bronze medals to still be a 0 points as late as at 2 hours and 30 minutes into the contest [8].
- There was a greater incidence of contestants *forfeiting* the contest: eight contestants forfeited the contest this year, while in onsite contests this number is usually zero or one.

To the best of our knowledge, there were no cheating attempts during the contest.

Technical aspects. The server fleet was composed by a central server and a variable number of worker machines, which were used to host the *cmsWorker* service of CMS. All machines were VMs hosted on Google Cloud. The central server hosted CMS, the service for collecting the screen streams from the contestants VMs, and the service for collecting reports from *oii-proctor*. In total, we spent around 100 euro for the whole cloud infrastructure. We also set up Grafana and Prometheus for monitoring the system: you can find the dashboards at [1] and [2].

The operating system chosen for the contestants' VMs was Ubuntu MATE, due to its commonality and relatively low resource requirements. When started, it would prompt a login screen which sent the inserted credential to the central server, which would send back a Wireguard (Donenfeld, 2017) configuration file: Wireguard was used to setup a secure connection between the VM and the server and the login served to keep track of which user was on which machine. Then, during the whole duration of the contest, a service on the VM would take a screenshot every 30 seconds and send it to the central server. This allowed us to further monitor the contestants. Also on all VMs an ssh daemon was installed, to allow us to enter the machine via the Wireguard tunnel.

oii-proctor is a program written in Rust (Matsakis and Klock, 2014) and distributed as a static binary for all three major operating systems, that served as an added measure to ensure some kind of proctoring for the host machine. After asking for a login to identify the user, *oii-proctor* collects information about the running processes to check if any browser is open and if the VirtualBox process is still alive, and send this information to the central server every 15 seconds. While doing these background jobs, it also puts up a bit of a *security theater* (Schneier, 2006): to keep contestants under the impression that *oii-proctor* is constantly active doing some kind of work, thus making a would-be-cheater contestant more aware of being monitored, *oii-proctor* constantly prints meaningless information at irregular intervals of time, masking when the actual checks are done.

5. Conclusions

In this paper we described how we moved online all the phases of the Italian Olympiads in Informatics. We hope that our experience, as well as the tools developed and the other ones mentioned, can provide to be helpful to other national or local programming contest organizers.

We provide a list of online resources after the references list, including github urls of the developed tools.

References

Amaroli, N., Audrito, G., Laura, L. (2018). Fostering informatics education through teams olympiad. In: *30th International Olympiad in Informatics, IOI 2018*, 12, pp. 133–146.

Casadei, G., Fadini, B., Vita, M.G. (2007). Italian Olympiads in Informatics. *Olympiads in Informatics*, 1, 24–30.

di Luigi, W. *et al.* (2018). Learning analytics in competitive programming training systems. In: *2018 22nd International Conference Information Visualisation (IV)*. IEEE, 321–325.

di Luigi, W. *et al.* (2016). oii-web: An interactive online programming contest training system. In: *Olympiads in Informatics*, 10, 195–205.

Donenfeld, J.A. (2017). WireGuard: Next generation Kernel network tunnel. In: *NDSS*.

Maggiolo, S., Mascellani, G. (2012). Introducing CMS: A contest management system. *Olympiads in Informatics*, 6, 86–99.

Matsakis, N.D., Klock, F.S. (2014). The rust language. *ACM SIGAda Ada Letters*, 34(3), 103–104.

Prechelt, L., Phlippsen, M. (2000). JPlag: Finding plagiarisms among a set of programs. In: 2000.

Schneier, B. (2006). *Beyond fear: Thinking Sensibly about Security in an Uncertain World*. Springer Science & Business Media.

Online resources

[1] *Dashboard with CMS's metrics*. URL:
`https://snapshot.raintank.io/dashboard/snapshot/0J29w0KruEiymy6zV30beX98aRG6njiX`

[2] *Dashboard with System's metrics for CMS*. URL:
`https://snapshot.raintank.io/dashboard/snapshot/1s1wQCSYPgWdQe9f6sAeYZns5Ln1y6e5`

[3] *Dashboard with System's metrics for Terry*. URL:
`https://snapshot.raintank.io/dashboard/snapshot/kPnzTPMNAIAmOHuubdds2ltUmdJ3Q9Xc`

[4] *Dashboard with Terry's metrics*. URL:
`https://snapshot.raintank.io/dashboard/snapshot/Uw7uECvHWCbYjoNicT1DSn4ZKhdbamSd`

[5] *OII-Proctor: a portable proctoring script*. A public URL is not available for security purposes. You can ask

the source code at this email address: `info@olimpiadi-informatica.it`

[6] *randomTeX: a quiz randomizer for LaTeX*. URL: `https://github.com/olimpiadi-informatica/randomtex`

[7] *Starplag: a tool for finding the similarities between two source files*. URL: `https://github.com/olimpiadi-informatica/starplag`

[8] *Statistics about the Italian National Phase*. URL: `https://stats.olinfo.it`

[9] *Terry: a very customizable "Google Code Jam" clone useful for holding programming contests*. URL: `https://github.com/algorithm-ninja/terry`

G. Audrito is involved in the training of the Italian team for the IOI since 2006, and since 2013 is the team leader of the Italian team. Since 2014 he has been coordinating the scientific preparation of the OIS and of the first edition of the IIOT. He got a Ph.D. in Mathematics in the University of Turin, and currently works as a Junior Lecturer in the University of Turin.

W. Di Luigi is involved in the training of the Italian team for the IOI since 2013. He holds a Master degree in Computer Science and Engineering from Politecnico di Milano, and currently works as a Software Developer.

L. Laura is currently the president of the organizing committee of the Italian Olympiads in Informatics that he joined in 2012; previously, since 2007, he was involved in the training of the Italian team for the IOI. He is Associate Professor of Theoretical Computer Science in Uninettuno university.

E. Morassutto is involved in the training of the Italian team for the IOI since 2016, with a particular focus on the technical aspects of the contests. Since 2017 he's involved in the OIS and IIOT as scientific and technical committee member. He got a Bachelor's degree in Computer Science and Engineering at Politecnico di Milano and he's enrolled in the same Master's degree course.

D. Ostuni is involved in the training of the Italian team for the IOI since 2015, as well as focusing on the technical aspects of the contests organization. He got a Master's degree in Computer Science at the University of Milan and he's currently pursuing a Ph.D. in Computer Science at the University of Verona. He firmly believes that $\tau > \pi$.

Olympiads in Informatics, 2021 Vol. 15, 13–21
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.02

Self-Generated Figures in Sequence Processing

David GINAT
Tel-Aviv University, Science Education Department
Ramat Aviv, Tel-Aviv, Israel 69978
e-mail: ginat@post.tau.ac.il

Abstract. A figure may convey an idea, an argument and even a proof, sometimes better than words. It may also elicit an idea, an argument and a proof. In problem solving, a figure may give a "feel" of a problem. A self-generated figure may help getting insight, or serve as a means for representing one's inner associations, or mental model of the problem. This paper presents self-generated figures in algorithmic problem solving. Students of our IOI advanced stage demonstrated constructive utilization of self-generated figures in solving challenging sequence processing tasks. The figures elicited associations of hidden patterns, whose recognition yielded elegant and efficient algorithmic solutions. We advocate the application and examination of self-generated figures in algorithmic problem solving.

Keywords: algorithmic problem solving, figure drawing.

1. Introduction

One of the essential problem solving heuristics is that of drawing a figure (Polya, 1945; Schoenfeld, 1985; Van Meter & Garner, 2005). Figure drawing is apparent in geometrical problem solving (e.g., Nunokawa, 2004), in the solution of word problems (e.g., Van Essen & Hamaker, 1990), in physics problem solving (e.g., Maries & Chandralekha, 2017), and more. In geometry, drawing involves dynamic examination and manipulation of the givens; in word problems it includes modelling, or representation of relations between objects; and in physics it involves both. These facets are evident in problem solving in computer science and algorithmics as well. For example, in OOP figures are used for modelling, or representing the "world", while in graph algorithms they are used for examining different execution progression scenarios. Algorithmics also involves the tool of visualization, which offers the display of entity values during execution of computer programs (Sorva *et al.*, 2013).

Drawing of figures, and visualization offer means for analysis and comprehension of given problems and their solutions. Yet, current algorithms textbooks do not advocate the drawing of self-generated figures during the process of problem solving. Textbooks use a lot of illustrations – of data structures, relations, functions, generic algorithmic ideas (e.g., binary search), and more. They exemplify and advocate heuristics such as problem decomposition, divide and conquer, backward reasoning, inductive reasoning, and more (e.g., Cormen *et al.* 1990), but do not underline the invocation of the heuristic of visual analysis of a given task. Visual analysis may be very helpful. In our experience with problem solvers, upon IOI training at all level, not many invoked this heuristic, but those who did invoke it often demonstrated its illuminating role in attaining sound solutions. Some of these students' self-generated drawings yielded elegant solutions together with their justifications. The drawings elicited constructive connections between elements, helped consolidating ideas and observations, and enabled external representation of inner mental models (Johnson-Laird, 1980) of posed problems.

In the next section we display illustrations of these phenomena, in the sub-domain of sequence processing. Sequences (and lists) involve a variety of fundamental tasks, from max computation, through searching and sorting, to string matching and more (e.g., Manber, 1986). The illustrations illuminate visual analyses that were offered by students. In the last section we discuss these findings of self-generated drawings, and argue the heuristic's relevance for teaching, displaying it to students, and encouraging them to employ it.

2. Visual Analysis Illustrations

This section shows three different scenarios of visual associations demonstrated by students. Each scenario involves a task that we posed during the third, advanced stage of our national activity towards the IOI. About 30 students reach this stage every year, and its early tasks require basic algorithmic knowledge, limited to 1D arrays, functions, recursion, searching and sorting, and complexity measures. Yet, the problem solving is not elementary. On the way to this stage, as well as in its early practices, students attempt algorithmic challenges whose solutions required capitalization on hidden, unfolded patterns. The tutors' challenge is to pose tasks that require little knowledge, but involve challenging problem solving for that stage. Diverse sequence processing tasks are suitable. The illustrations below include three such tasks.

The following task was one (the easier) of the six tasks of IOI 2005 in Poland. The phrasing here is shorter and less formal.

Mean Sequence. Given an increasing sequence of N positive integers, output the <u>number</u> of mean-sequences of length N+1 of the given sequence; where a *mean-sequence* is a sequence of non-decreasing integers, such that the mean of the i-th and the i+1-th integers is the value of the i-th element in the input sequence.

<u>Example</u>: For the input **4 9 16 21 23** the output should be **3**, as there are 3 mean-sequences (of 6-integers) that fulfill the required condition: **2 6 12 20 22 24** and **3 5 13 19 23 23** and **1 7 11 21 21 25**. Notice that every integer in the input se-

quence is the mean of two corresponding integers in each of the above sequences. Also notice, that a sequence starting with **0** may not be a valid mean-sequence, as the sequence will be: **0 8 10 22 20 26**, which is decreasing between the 4-th and the 5-th integers. For a similar reason, sequences starting with **-1**, or **4**, may also not be valid mean-sequences.

A first glace over the task yields several observations.

- Once an integer is chosen as the starting point, the rest of the sequence is determined.
- The output may not exceed by more than 1 the difference between the "closest" two integers in the input. (In the example of the problem statement, the two closest integers are **21 23**, thus the output may not exceed **3**.)
- A hasty conclusion from the previous observation may lead to the assertion that the value mentioned in the observation will be the output. Yet this may not always be the case; e.g, for the input sequence **4 6 10 20 22** the output should be **0** (and not **3**).

Following the first observation, problem solvers who follow a brute-force approach often seek a solution based on examining separately each sequence yielded from a starting point that enables the first two mean sequence points (the point before the first input value and its successive mean sequence point). This solution requires O(N×D) time and O(N) space, where D is the difference between the first two input points.

Other problem solvers regard the latter solution inefficient and seek a hidden pattern on which to capitalize. One such problem solver, whom we interviewed, conjectured that the starting points of legal sequences may be adjacent to one another, and form a range of consecutive integers. In order to examine his conjecture, he had a visual association. His idea was to start with a range of size D+1 and view each of the input integers as a *hinge* for *flipping* the initial range "forward", through the input points while possibly chopping it. He viewed the process of "flipping" the range, as a *"rod shrinking through the input values"*, and examined his idea with: **4 9 16 21 27 32 37**.

The visualization in Fig. 1 displays an elegant algorithmic idea, and informally also shows its justification. Although the figure only shows an example, and does not display formal notations, one may extract an argument of correctness from the figure. In a sense, it offers a hand-in-hand design + justification, and yields the following task solution.

> *The output is the size of the range remaining from "passing" the initial range of successive candidates through the input sequence, in a manner of flipping a "shrinking rod" through a series of hinges.*

The solution scheme enables an on-the-fly computation. All in all, the "flipping" process requires O(N) time and O(1) space. The computation visualized in Fig. 1 displays a scenario that represents the students' inner mental model of the task solution. The figure elegantly expresses an intuitive, elegant idea, without any notations.

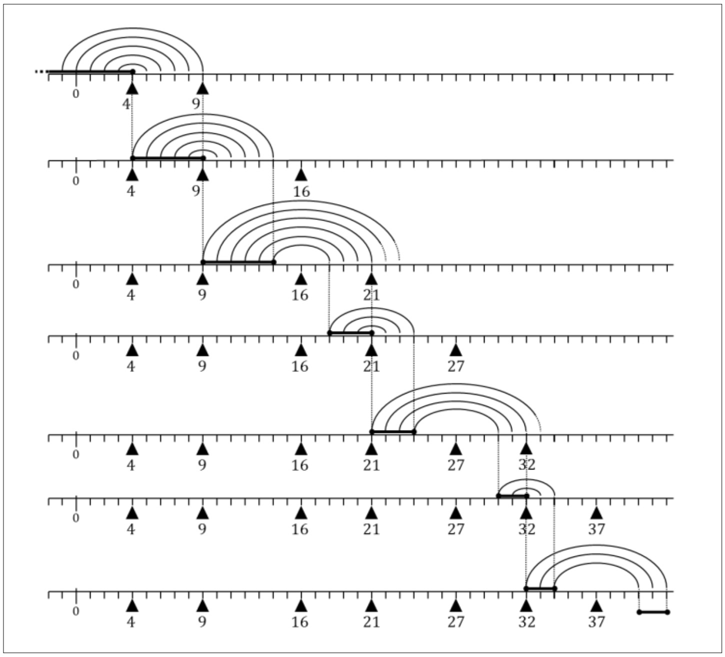


Fig. 1. Visual association of *flipping while chopping* the initial range of starting-points through the input sequence.

* * *

The next illustration involves visualization related to ordering. Ordering is a fundamental notion in algorithmics. While the natural tendency may be to simply examine integers, a figure that displays heights of values may help reaching beneficial associations.

Widest Inversion. Given a sequence of N different integers, output the widest inversion in the sequence, which is the maximal distance between two unordered integers, i.e., the larger of them is to the left of the smaller.

Example: For the input **4 3 8 9 1 6 10 7** the output should be **5**, due to the distance between the **8** and the **7**. Notice that there are many inversions in this sequence, e.g., the inversions of **4** and **1**, of **4** and **3**, of **3** and **1**, of **9** and **7**, and more.

(We first introduced this task in Ginat (2008), with diverse correct and incorrect solutions. Here, we examine it with respect to the relevance of drawing a figure.) One of the problem solvers of this task chose to examine it with several examples, for which she drew figures of bars. Below is one of the figures, for an input which is a permutation of the integers 1..11.

The student indicated that in the beginning she examined the brute-force solution, which computes the widest inversion for each input value as a left-end of an inversion. Obviously, the computation is inefficient, as it requires repeated "passes" over the input. In addition, the student mentioned that it was difficult for her to gain insight into the task

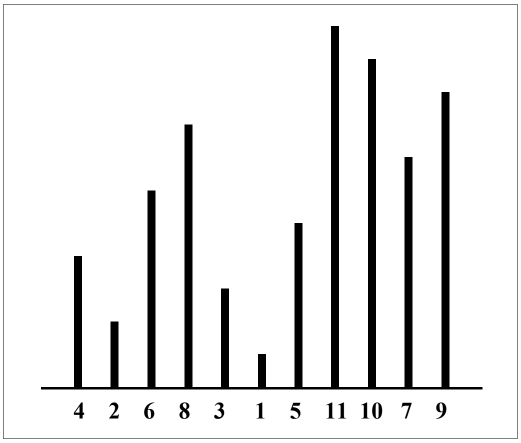


Fig. 2. Bars that represent the input values by height.

by repeatedly looking at series of integers. She sought a representation of the task that will be more illuminating. At first, she tried to draw lines between inversion ends in the list of integers, but decided that these lines do not clarify the "blurred" picture. Her next attempt was to draw a figure of bars, whose heights correspond to the input values, as displayed in Fig. 2. This representation helped her realize several observations, which "popped from the figure", regarding left-ends and right-ends of inversions.

- A value v may be in several (even many) inversions as a left-end. The only relevant value to consider as a right-end for v is the rightmost value u smaller than v. Thus, u must be smaller than all the values on its right.
- A similar observation is relevant for a value v as a right-end.
- The above observations imply that a value may be a **candidate for a right-end** of the widest inversion only if it is smaller than all the values on its right.
- Similarly, a value may be a **candidate for a left-end** of the widest inversion only if it is larger than all the values on its left.
- In the above figure, the values **1**, **5**, **7**, **9** may be candidates for a right-end of the widest inversion; and the values **4**, **6**, **8**, **11** may be candidates for a left-end.
- The two lists of candidates must be **in increasing order**, due to the properties of candidates. (Question: May a particular value be in both lists of candidates?)

Following the above observations, an O(N) time and space solution will first generate the two lists of candidates, and then capitalize on their increasing characteristic and "run" concurrently over these lists from left to right, while finding for each left-end candidate its corresponding right-end match.

All in all, the visual analysis enabled illuminating observations, that stemmed from looking at the bar heights and noticing the characteristics of left-ends, right-ends, and increasing candidate lists. In our experience, although these characteristics are simply phrased, problem solvers often struggle and do not recognize them while "going back and forth" over the sequence of integers. The drawn figure assisted in revealing them, and invoked constructive associations.

* * *

The last illustration involves visualization related to the recognition of a hidden sequence property on which to capitalize.

Circular Fence. Given a sequence of N positive integers, each describing the height of a column of a fence, in terms of its number of bricks, output the minimal number of blocks of bricks that should be transferred between adjacent columns in order to level the fence. The total number of bricks in the fence is **3N**, and the fence is **circular** (the N-th column is adjacent to the the 1-st column). A transfer of a block of bricks may involve any positive number of bricks, which will be moved either to the left or to the right of the column from which they are taken.

Example: For the input **5 3 6 2 1 1** the output should be **3**, due to a transfer of a block of 2 bricks from the column of **5** to the column of **1** cyclically on its left, and two more transfers of bricks – 3 bricks from **6** to **2**, and then 2 bricks from **2** to **1**.

One may first attempt the task for a **linear**, non-circular **fence**, with a left end and a right end. A single "pass" over the input from left to right will yield the desired output. The input will be accumulated, and for every i, 1 ≤ i ≤ N, if the total number of bricks accumulated up to (including) the i-th column does not equal i×N, then the total number of block-transfers will be increased by 1. The latter operative description implies the following declarative observation, which relates to the places of no block transfers.

*Let S be the number of autonomous sub-sequences. A sub-sequence is regarded **autonomous** if its total num of bricks is 3 times its num of columns, and no prefix of it is has this property (of 3 times the num of columns). The output is N-S.*

Notice that if the fence in the example of the task statement is regarded linear, then the whole fence is one autonomous sequence. It cannot be partitioned into smaller autonomous parts. Since there are 6 columns in the fence, a total of 5 block transfers are required. Had the fence been circular, it could be partitioned into 3 autonomous sub-sequences, and only 3 transfers were required.

The reader may assess the correctness of the declarative observation above, for a linear fence. Notice that in the case of a linear fence there is only one partition possible, since there are given left end and right end. In the case of a circular fence there may be a variety of possibilities for partition. We need to find the best partition, i.e., the partition that yields as many autonomous sub-sequences as possible.

One way of seeking the best partition is to examine N different cases of a linear fence – the case of column-1 as the left end, then column-2 as the left end, then column-3, and so on. However, this solution is "expensive" time-wise. Its time complexity is O(N^2).

Two students who sought a more efficient solution drew figures, which elicited their associations to a more efficient solution. One of them considered the following input of 11 columns **5 2 7 1 1 2 4 6 1 2 2** and examined the accumulated value of remaining bricks while levelling the fence on-the-fly, **2 1 5 3 1 0 1 4 2 1 0** (starting

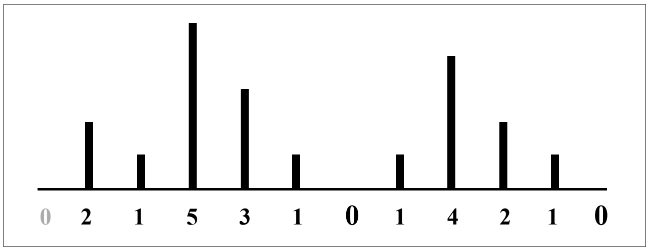


Fig. 3. The accumulated values while levelling the fence on-the-fly, starting at column-1.

from column-1). The leftmost **2** in the latter sequence indicates the number of bricks left after leveling column-1, the following **1** indicates the number of bricks left after leveling the first two columns, and so on. The last value in that sequence must be **0**, as the total number of bricks is 3N. Notice that for other inputs, the latter sequence could include negative values, which indicate deficits that require transfers of bricks to the left, from columns ahead. The student generated Fig. 3.

Fig. 3 shows that when the left-end is column-1, two **0**'s are generated, indicating 2 autonomous sub-sequences. When the student looked at the bars of the figure, he noticed that **1** appears 4 times. If he could "turn" the **1**'s to be **0**'s, he would have had 4 autonomous sub-sequences! He realized that this can be obtained by starting the levelling from the 3-rd position with **0** accumulated bricks, as shown in Fig. 4.

The student noticed that he could also start from the 6-th, 8-th, or 11-th column, and obtain the same result of 4 autonomous sub-sequences. He noticed that no matter where one starts the cycle, the sequence of accumulation-while-levelling numbers will be a shift of the original sequence in Fig. 3. The difference will only be in the location of the X-axis of the figure.

All in all, he realized that he may generate the accumulation sequence just once, starting from column-1; and then find the number of appearances of the value S that appears the maximal number of times. The output will be N-S. The computation of S requires O(NlogN) time, a considerable improvement of the initial O(N^2) solution. Careful look at the initially drawn figure yielded an observation of the bar height that appeared a maximal number of times, and elicited the association of adjusting the X-axis, so that this height will be the new 0.

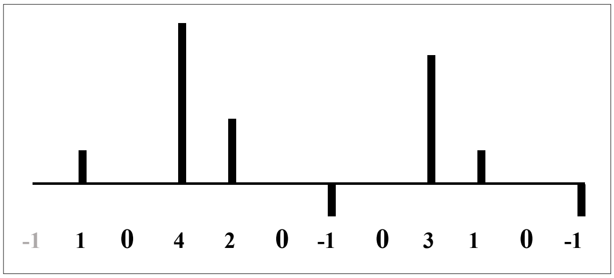


Fig. 4. The accumulated values while levelling the fence on-the-fly, starting at column-3.

3. Discussion

Visualization is a powerful method that can help one become more familiar with a given situation. It may give a "feel" of a posed problem and serve as a means for making explicit a detail or a picture in our mind. It may elicit constructive associations and connections between relevant elements. It may yield ideas. It may assist in reaching a plausible solution, as well as justifying it. A picture is sometimes worth a thousand words.

We have shown three different occasions of self-generated figures, that considerably assisted algorithmic problem solvers. Drawn figures elicited constructive associations and encapsulated mental models of posed problems. In the solution of the first task, the figure drawn by the problem solver helped him explicitly represent his mental model of the problem, with elements that involved a metaphor of rod flipping steps. His drawn visual scenario yielded not only the algorithmic solution, but also its intuitive justification, without formal mathematical notations.

The solution of the second task involved visual display of integers as bars of different heights. The visual display helped seeing ordering characteristics, especially with respect to left-end and right-end properties of inversions. The solution of the third task involved bars of integers as well, but the elicited associations were related to the recognition of multiple appearances of equal elements. The drawn figure yielded a constructive up/down shift of the X-axis of the represented values, and led to an elegant and efficient solution of sequence partitioning. The shift manipulations encapsulated not only the solution but also its justification.

Self-generated figures as a heuristic means in problem solving are studied in mathematics (e.g., Nunokawa, 2004), physics (e.g., Maries & Chandralekha, 2017), and additional domains, but not in computer science (to the best of our knowledge). In computer science, figures and visualizations mostly appear as teaching aids and computational-tracing means. Yet, self-generated figures may be most beneficial for problem solvers, as we have seen here. We believe that this problem solving heuristic should be elaborated, exemplified, and studied in the teaching of algorithmics, at all levels, including the Olympiad level. The enriching potential of figures may elicit and communicate associations and ideas informally, without words, but in a clear, elegant, and convincing manner.

References

Cormen, T.H., Leiserson, C.E., Rivest, R.L. (1990). *Introduction to Algorithms*. MIT Press.
Ginat, D. (2008). Learning from wrong and creative algorithm design. In: *Proc of the 40th ACM Computer Science Education Symposium – SIGCSE*. ACM Press, pp. 26–30.
Johnson-Laird, P.N. (1980). Mental models in cognitive science. *Cognitive Scienc*e, 4, 71–115.
Larkin, J.H., Simon, H.A. (1987). Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11, 65–99.
Manber, U. (1986). *Introduction to Algorithms: A Creative Approach*. Addison Wesley.
Maries, A., Chandralekha, S. (2017). Do students benefit from drawing productive diagrams themselves while solving introductory physics problems? The case of two electrostatic problems. *European Journal of Physics*, 39, 1–18.
Nunokawa, K. (2004). Solvers' making of drawings in mathematical problem solving and their understanding of the problem situations. *International Journal of Mathematical Education in Science and Technology*, 35, 173–183.
Polya, G. (1945). *How to Solve it*. Princeton University Press.
Schoenfeld, A. H. (1985). *Mathematical Problem Solving*. Academic Press.
Sorva, J., Karavirta, V., Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 15, 1–64.
Van Essen, G., Hamaker, C. (1990). Using self-generated drawings to solve arithmetic word problems. *The Journal of Educational Research*, 83, 301–312.
Van Meter, P., Garner, J. (2005). The promise and practice of self-generated drawing: literature review and synthesis. *Educational Psychology Reviews*, 17, 285–325.

D. Ginat – headed the Israel IOI project in the years 1997–2019. He is the head of the Computer Science Group in the Science Education Department at Tel-Aviv University. His PhD is in the Computer Science domains of distributed algorithms and amortized analysis. His current research is in Computer Science and Mathematics Education, with particular focus on various aspects of problem solving and learning from mistakes.

Olympiads in Informatics, 2021 Vol. 15, 23–36
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.03

On Using Real-Time and Post-Contest Data to Improve the Contest Organization, Technical/Scientific Procedures and Build an Efficient Contestant Preparation Strategy

Jamaladdin HASANOV, Habil GADIRLI, Aydin BAGIYEV
ADA University, School of IT and Engineering
Ahmadbey Aghaoglu str. 61, 1008 Baku, Azerbaijan
Email: jhasanov@ada.edu.az; hgadirli2019@ada.edu.az; abaghiyev@ada.edu.az

Abstract. Nowadays, the coaches of various sports disciplines use analytical tools to process the game data, analyze the behavior of their team, individual players, and build efficient strategies based on the strength and weaknesses of the competitors. Considering its global scope, time limitation, complexity of tasks, ranking, and medals, IOI can also be considered as an intellectual sports contest. With the direct relation to Information Technology and sports, the statistics of the IOI contests need to be analyzed to deliver better results, initate changes in the preparation strategy and increase the quality of the event as well. This paper provides a statistical analysis based on the last on-site IOI and shares insights on each of them.

Keywords: CMS, data analysis, contestant, organization.

1. Introduction

International Olympiad in Informatics is a global algorithmic programming contest that involves school-aged contestants worldwide, organized as teams, representing their countries. According to the statistics of the last 5 years (IOI Statistics, 2020a), more than 300 contestants from at least 80 countries have joined the contest each year. The final ranking of the contestants after a 2-day competition is highlighted with the corresponding medal’s color code on the results page of every contest year (IOI Statistics, 2020b). The number of countries, contestants, team members, guests, and standings are the basic statistics shared with the public by the International Committee of the IOI.

More detailed statistics prepared by the Technical and Scientific committees are shared with the internal community:

- The International Technical Committee shares a summary on the technical issues, common problems and provides statistics on the language usage, task completion, and other contest-related information that is fetched from the contest database. These inputs help improve the infrastructure-related planning (contest environment, contestant laptop parameters, and so on), optimize procedures (of printing, translation, contests start/stop, extension rule, backup rules, etc.), and help Scientific Committee make adjustments to the rules (on adding/removing programming languages, modifying limitations, tasks types, changes in regulations and so on) if necessary.
- The International Scientific Committee runs a survey to get contestant and team leader responses. This survey usually covers the tasks (statements, difficulty of tasks, and subtasks), contest environment, online judge systems, programming language preference, and organizational questions. The survey result may also affect the procedures and regulations – during its presentation in the joint meeting of the committees, some common concerns and points may raise an internal discussion.

As stated previously, these inputs help better understand the strong/weak points and continuously improve the quality of the contest. According to the reports of the last years, more common and basic organizational problems are not encountered anymore, the contest environment, judging system, and procedures have been improved to eliminate the usage and capacity-related problems.

Despite its immense value on organizational planning, the mentioned reports do not cover the information that describes the contestant behavior, which might be interesting to the contestants and their team leaders for retrospective analysis and also provide Technical and Scientific teams with valuable information right on time. In recent years, there has been valuable research and applications of analytical methods and Machine Learning techniques to predict the contestant performance (Alnahhas and Mourtada, 2020) and analysis of the task difficulty (Fantozzi and Laura, 2020; Vegt and Schrijvers, 2019; Pankov and Kenzhaliyev, 2020). Some specialists have provided valuable pedagogical analysis of the key factors that may lead to contestant's success (Tsvetkova and Kiryukhin, 2020; Lodi, 2020). There have also been ideas on platform-based automatic analysis of the contestant activities in competitions and e-learning systems (Kostadinov *et al.*, 2018).

This paper introduces a report, based on the IOI 2019 data, which aims at understanding of the contestant behavior during the contest days. This report is the first phase of the "IOI data analysis" project supported by the grant, which provides an analysis of the Contest Management System (CMS, n. d.) database of the corresponding IOI archive.

The following chapter shares various statistical reports, provides authors' inference based on the data, and some recommendations derived from them.

2. Insights

The Contest Management System (CMS) is an open-source distributed contest management and grading system that was developed for the IOI 2012, hosted in Italy. Starting from 2017, the CMS system has been used consecutively in the 5 last IOI contests, which made it a potential nominee for the de-facto standard for the IOI. CMS stores all the task and contest data in a compact Postgres DB structure. CMS has decent documentation on the structure and installation of the system but the structure of the tables and their relationship are not fully described. As a part of this report, the structure of the DB and the purpose of the tables have been documented and shared in a public repository (GitHub, 2021a).

This chapter introduces the insights on IOI 2019 data, that start as general statistics and then specific targeted analysis, accompanied with the findings.

2.1. *Submission Dynamics*

On each day of the contest, the contestants receive 3 tasks and 5 hours of time to submit their solutions. The order in which they solve the tasks and frequency of their submissions might be different (which is analyzed in 2.3). The following report shows the number of submissions in a 30-minute interval (Fig. 1). Each diagram shows the number of total (blue line) and successful (dashed orange line) submissions. Any submission scored more than 0 is considered as successful. So the gap between the lines shows the submissions that did not score anything.

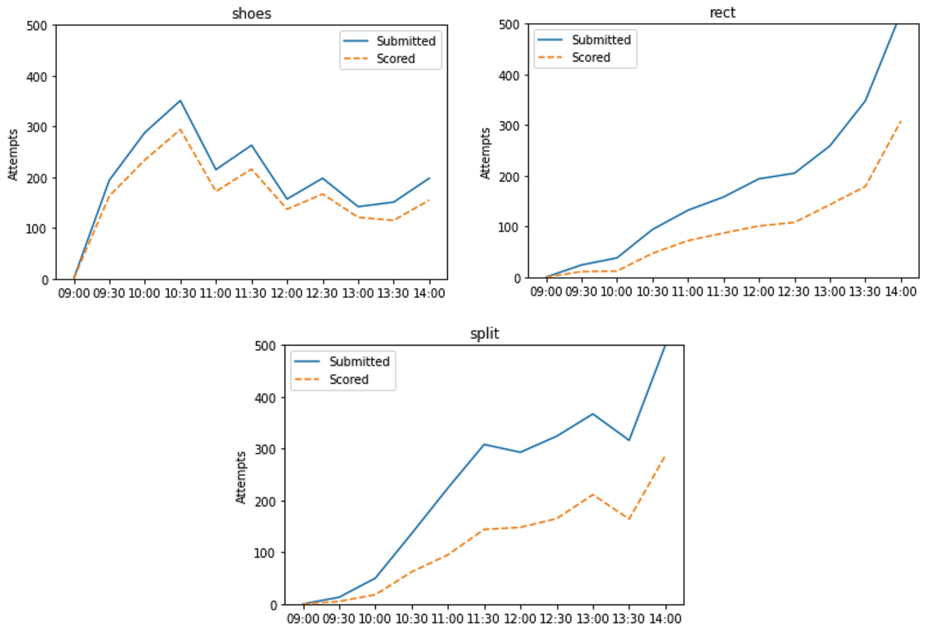
As seen from Day 1 results, "Shoes" was a comparably easy task based on a greedy algorithm and therefore gets a lot of submissions from the start. "Rect" has some increasing dynamics but "Split" seems challenging till the end.

Before the contest, the Scientific Committee (SC) of the IOI evaluates the complexity of the tasks and makes assumptions on solvability by the contestants (like 65% of the contestants may partially or completely solve it). Day 2 tasks have the same flavor – "Line" is easy, almost all the attempts have been graded (only less than 10% of the total contestants couldn't score anything), but the other two tasks were quite challenging for the contestants.

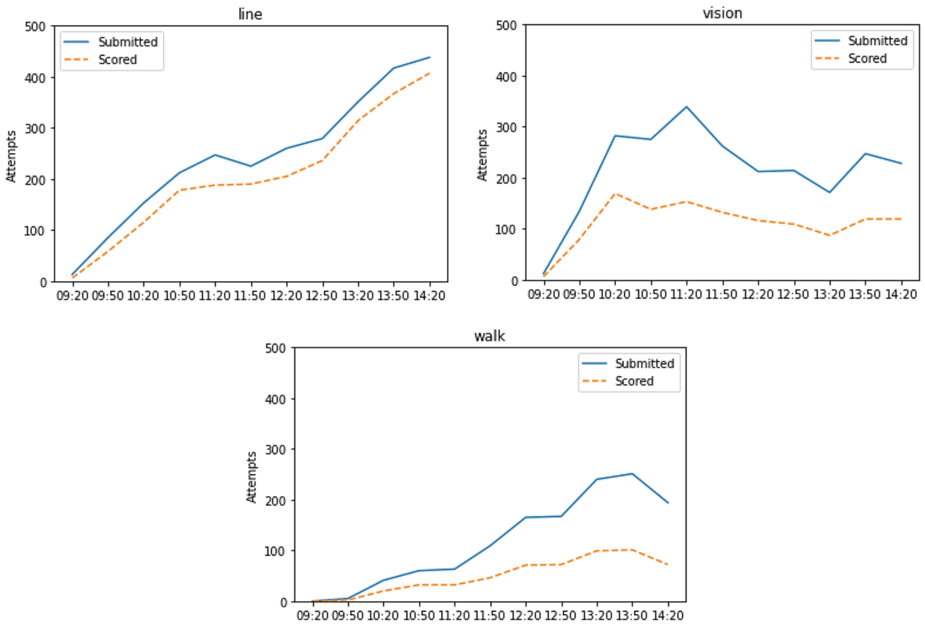
The mentioned insight might be useful for the real-time and post-contest analysis by the SC, and add value to the task selection process.

2.2. *Contestant Requests*

The IOI's scope is not limited only to the technical procedures – the administrative, logistic, and support services are also primary responsibilities of the host committees. During their 5-hour work, contestants use their contest environment to ask various ques-

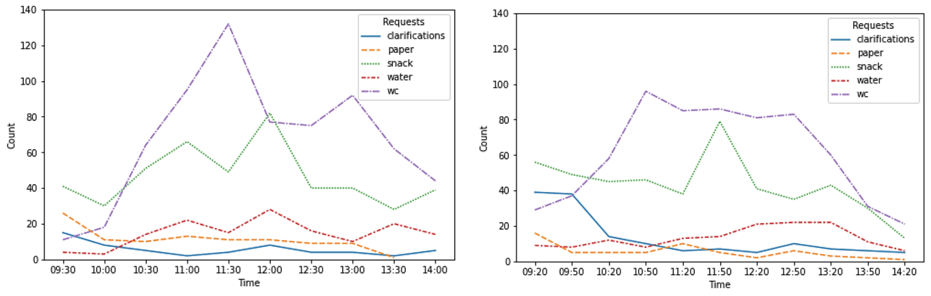


Day 1



Day 2

Fig. 1. The number of total and successful submissions grouped by tasks.



Day 1 Day 2

Fig. 2. Contestant requests and questions.

tions, starting from the clarifications to the request for food, beverage, and other supplies. During the IOI, this statistic is usually shared in a summarized way (what type of requests in total we had). Fig. 2. shows the number of the requests made by contestants every 30-minute, grouped by the request type.

In the IOI2019 version of the ContestWebServer, each question had a corresponding icon that helped contestants register their needs/questions/inquiries just by clicking on items and adding a small note. As a result, it was quite easy to group the results by categories. The "snack" category includes fruits (banana and apple), chocolate, and cupcake. The increase of the "snack" and "WC" requests at the beginning of day 2 is connected to the late start of the contest (in contrast with Day 1, the Day 2 statistics includes the contest's start time, which was 9:20. Due to delay on the 2nd day, the requests have started before the contest time. Therefore 9:20 shows the accumulation of the requests till that time.). As seen from the graphs of both days, the clarification requests are naturally high at the beginning of the contest – all technical and rule-related questions are clarified in the first 30–40 minutes of the contest. Also, there is an increase at the start point of the "clarifications", which was due to technical problems. As seen from the graphs, for both days, after 45 minutes from the start, the "clarifications" count starts to decline.

As seen from the graph, putting water bottles on the desks before the contest and adding a few empty A4 papers in the task envelopes reduced the request on these tasks. This improvement had been noted from the previous IOI observations – a systematic review of the request fulfillment statistics might increase the efficiency of the process.

During the contest time, HTC team members are the only staff that has a chance to closely observe the situation, see the patterns and share their thoughts and forecasts immediately with their coordinators or HTC chair. At the 2nd hour of the contest day 1 of IOI 2019, HTC team members shared with the HTC coordinators that there is an increasing queue to the toilets (as might be seen from Fig. 2. Day 1). In the beginning, it was thought that it might be normal (usually this peak is common between the 2nd and 3rd hours of the contest) but considering the number of allocated WC cabins in National Gymnastic Hall (which hosted events with 10,000 visitors), it was decided to double-check the case. According to the observation of the HTC members, the reason could

Table 1
Sequence analysis between the snack and WC requests

Original requests	The number of WC requests	
	Day 1	Day 2
Banana	70	-
Apple	-	34
Chocolate bar	58	50
Cupcake	22	16
Water	98	77

be the order of bananas and drinking water over it. As a result, it was decided to stop delivering bananas (replaced them with chocolate bars) and to replace the banana with the apple on day 2. As seen from Fig. 2. Day 2, the "WC requests" do not cross the 100 requests and distribution seems quite normal.

Now in the retrospective analysis, we can see the proof of the behavior. Table 1 shows the number of WC requests made within the 45 minutes after each snack and water-related request. As seen from the Day 1 numbers, the order of 'Banana' and 'Water' seem to have a correlation with the 'WC' requests. There is a high number for 'Chocolate bar' as well, but when analyzed, it was realized that mainly they are either ordered or consumed with water.

If such a report had been used in a real contest time, the mentioned incident could have been detected as a pattern way before its peak time. Similar to technical alarm and monitoring systems, HTC could easily use pre-defined thresholds and rules to see the irregular or abnormal patterns and involve HTC members in the detailed investigation.

2.3. *Switching Between the Tasks*

Knowledge for sure plays an important role in a contestant's success. But problem-solving and evaluation skills definitely cannot be ignored as well. A few scenarios with the various outcomes might be possible:

- A contestant may start with the most difficult task, spend the majority of time on it and exhaust himself/herself without leaving no chance to solve the others.
- A contestant may decide to spend a fixed amount of time (small, just enough to analyze the problem) on each of the problems in the beginning, evaluate the complexity, select the easiest one and solve one at a time.
- A contestant may decide to work on the hardest task first, such that it requires more concentration, and then spend the remaining part of the contest to the simple tasks.
- Same as in the previous option, but a contestant may decide to use an iterative approach to spend a fixed amount of time on each task and this way focus on the incremental total score, rather than the finalization of the particular task.

The selection of the right strategy for the contestant is a primary job of the coach and the decision on the particular strategy shall consider many factors, including:

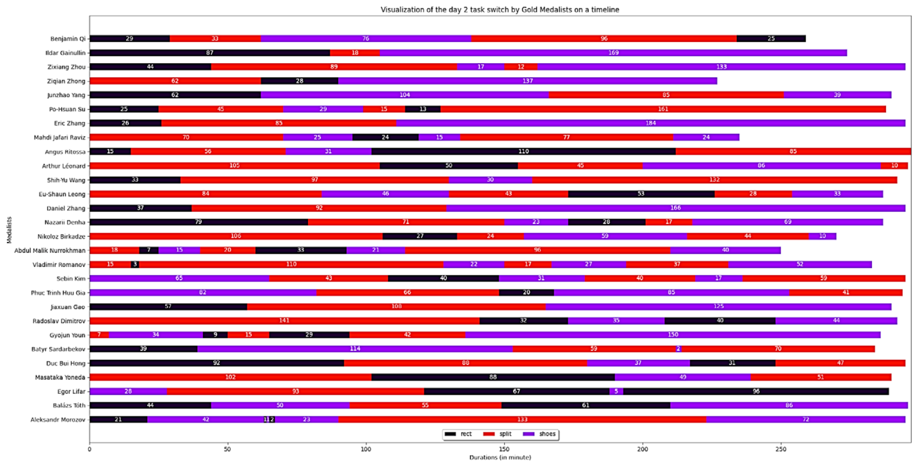
- The continuous concentration time of the contestant.
- The total amount of time, a contestant can be productive (it is in no way all 5 hours of the contest day for everybody – there have been contestants who decided to take a nap during the contest).
- The knowledge of the contestant on different problem types.
- The cost (of time and attention) of switching from one task to another.

The teams that regularly deliver good results definitely have well-rehearsed strategies. The teams that usually have lower standings or are aiming to achieve even better results need to analyze the behavior of the successful contestants and infer knowledge from it. The ranking page of CMS allows to see the dynamics of the submissions for the particular task of each contestant but the analysis of each might be quite a tedious job. In this paper, we consolidate this data and deliver statistics on task solving. To demonstrate a successful model, we provide the list of IOI 2019 gold medalists (on the Y-axis) with their work progress throughout the contest (X-axis) (Fig. 3). Each color in the graph corresponds to the particular task and the number on it shows the minutes spent on that task during that work. Each bar in this graph may include several submission attempts. As seen from the graph, some contestants haven't made that many task switches (On day 1, Ildar, Ziqian, and Eric have worked on one task at a time, but Benjamin and Zixiang have returned to the previous tasks twice), whereas others had the mentioned iterative approach. The same approach is followed the next day, which means the approach was consistent and rational.

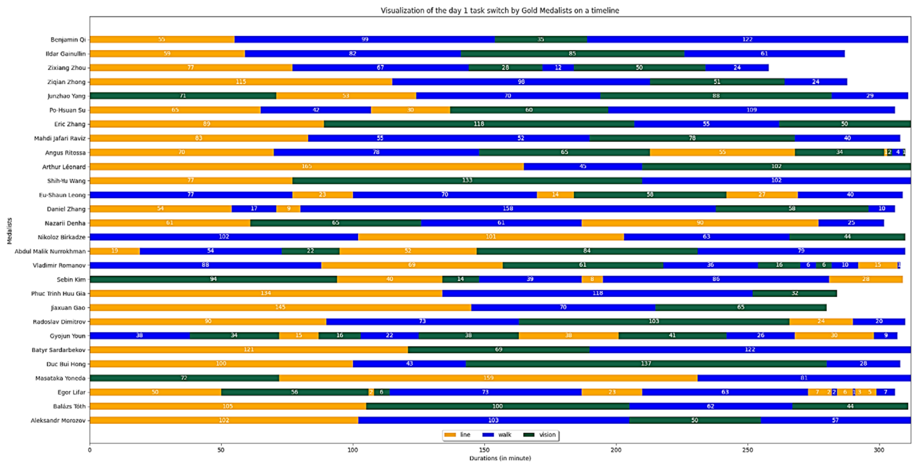
Another interesting insight from this graph is, the majority of the gold medalists did not choose "shoes" as a first task. Our first graph in Fig. 1. shows that during the first 30 minutes, there have been almost 200 submissions on "shoes". By analyzing the current graph, we can say that future "gold" medalists have been a very small portion of these submissions. As seen from the graph, "shoes" were not the second target of the "golds", either. We may infer two possible hypotheses from here: either the "golds" have started to work on the tasks in the order they got them from the envelope ("rect", "split" and "shoes" – there is no fact or record on the actual order of tasks) or there is a common strategy of successful contestants – start with the hardest and then do the easy tasks.

On the contrary, on day 2, they start with the "lines" problem, which was mentioned as "easy" in the previous analysis. This fact leads to the first option, which states the stronger contestants probably approach the tasks by their order in the envelope. Even if this is the strategy, by the similar pattern of the colors, it is discernible that they have a steady pace on problem-solving. As mentioned earlier, the selection of the first task is not the unique characteristic of the contestant – there is a particular pattern of switching (or not doing so) from one task to the previous tasks.

The knowledge cannot be gained only on the top positive and limited set of information, so to see the impact of the task-switching on the results, we analyzed the distribution of the task switches for all contestants, grouping them by the medals (and also not getting one). By the medal we mean the final reward of the contestant – it



Day 1



Day 2

Fig. 3. Gold medalists (sorted by standing) working on the tasks.

shall not be confused with the current standing. Fig. 4 shows the distribution of the amount of time spent on each task for each contestant category. The outliers with the minimum values (falling below Q1-1.5*IQR) have been removed from the data set. These are mainly the erroneous or random behavior (like working on a task for a few minutes, submitting and switching back to another task), that is not a subject of our analysis. On the contrary, the outliers with the maximum values are definitely valuable for the analysis.

As seen from Fig. 4, on day 1, almost 50% of the gold medalists spent from 30 to 70 minutes of time per task. The average time spent on a task in this category is around

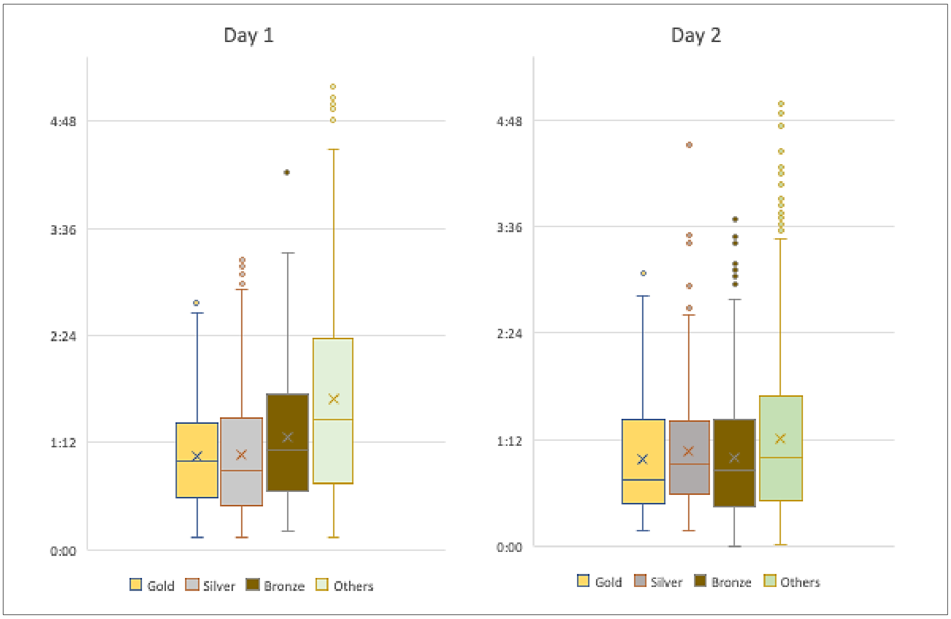


Fig. 4. Time spent on a task.

1 hour. It might be considered as a good time frame to keep the focus on a task. The silver medalists have the same average but with a wider variance and more outliers, that fall behind 2.5 hours bar. Despite silver medalists, the distribution of the bronze medalists does not share the common average with the "golds" and is shifted up. The average time spent on a task by the non-medalist category stands beyond 75% of the gold and silver medalists. 25% of non-medalists have spent more than half of the contest time on a single task and some contestants (outliers) have totally spent the contest time solving one task.

It seems the lessons learned (analyzed by the coaches or contestants themselves) applied to day 2:

- Gold medalists still had the same variance but some long attempts have been replaced with the small iterations.
- Silver medalists are still organized and follow the same plan. Some more outliers might be observed – contestants know their standings after day1 and work on the tasks that they are sure to score at.
- Bronze medalists show better performance – almost repeat the distribution of "golds". To secure their place for a medal, they need to be rational – not to spend time on a task, if they cannot succeed.

In Fig. 5, we show another distribution that is connected to the previous one – the number of repetitive switches made by each contestant. The number of repetitive switches is counted as the total number of switches from one task to another (including the first one) minus 3 (the number of tasks per day) – working on the tasks themselves for the

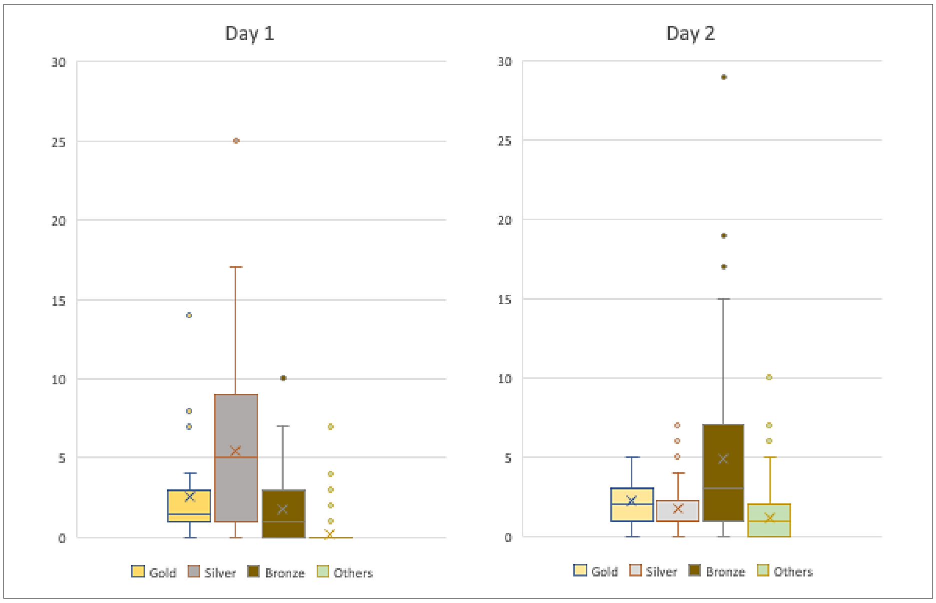


Fig. 5. The number of task switches.

first time is not counted as a switch. So in Fig. 3 day 2, Benjamin and Ildar both have made 1 repetitive switch.

Fig. 5. shows the number of switches which kind of is an addition to the previous graphs. As seen from day 1 graphs, despite the similar distribution of spent average time, “silvers” had way more task switches than “golds”. On both days, gold medalists make no more than 5 switches, except a few cases shown as outliers for the day 1 report. On the contrary, “bronzes” had fewer task switches, and contestants who did not get any medal almost had no switch on day 1.

The lessons learned from day 1 had decreased the task switch count for “silvers”. As explained for the task duration case, the “bronzes” started to score as much as possible by leaving the tasks, where they had no progress. These dynamics have touched the other contestants who did not receive any medal (as a result) – they stopped getting stuck at one or two tasks and started to try to have an iterative way.

According to the insights, the problem-solving process can be set as the following algorithm:

A1. Spend time to understand task X (say, the next one you get from the envelope). If the solution is obvious, move to A2. Otherwise, move to step A1.

A2. Start working on a task. Do you get positive/promising scoring?

- If it has been more than Y minutes, then move to A1, Otherwise, keep working.
- else, move to A1.

As mentioned earlier, the “Y minutes” value should be identified and set individually for each contestant by the coach.

2.4. *A Team Performance*

Teams that are associated with contestants' countries are usually introduced once in the Opening Ceremony of the event and occasionally mentioned in the cultural night events. Although IOI considers only individual contestants in the ranking and medal list, the medalists usually demonstrate their link to the particular country during the award ceremony. There is something more than just patriotic feelings in team recognition – a victory of the contestant is the victory of the coach and the system (educational, Olympiad movement in that country, the materials written and taught, trials, selection, and many others) that are not mentioned or noted as a part of the event. Regular country reports may include brief statistics about the country statistics, but it does not go deeper into the team performance analysis. One of the recent works provided an interesting analysis of the countries' performance in Science Olympiads, focused mainly on the particular regions (Jovanov *et al.*, 2018).

Taking this opportunity, we decided to show another report that shows the success of the teams according to their performance in IOI 2019 (Fig. 6). The graph shows the list of teams that owned at least one medal and a range of absolute scores that were made by their members. Teams with a single medalist have a single short vertical line. The teams are sorted from left to right by the decreasing order of the maximum team score. There are also three horizontal lines correspondingly showing the border for the "gold", "silver" and "bronze" medals scores.

As seen from Fig. 6, this graph resembles that ranking order, where team "USA" leads with Benjamin's highest absolute score 547.09. This report might help infer some more information, but the order of teams does not say anything about the performance of teams: although team "USA" got the highest score, the adjacent team "RUS" has better

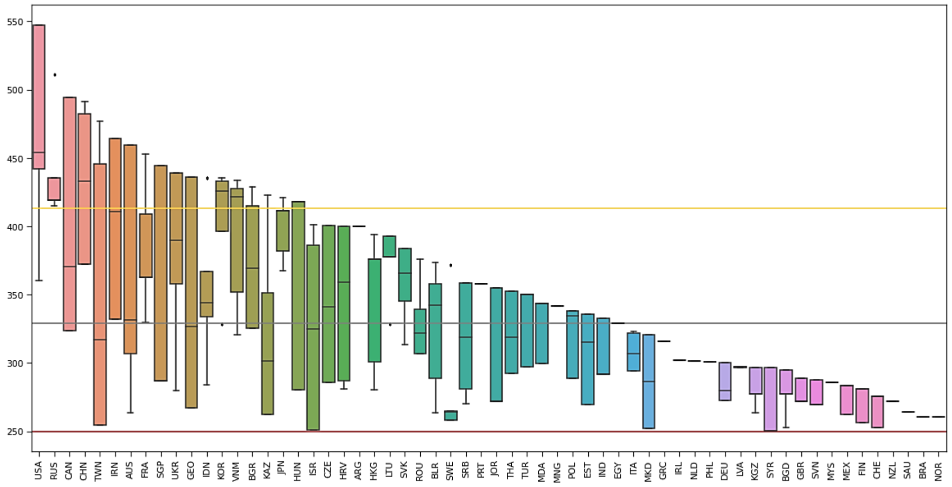


Fig. 6. Teams with medals and their score ranges (sorted by maximum value).

results – all 4 contestants got “gold” medals. Or team “TWN” in total scored more than team “JPN” (477.14 vs 420.89) but the range of scores are way much wider ([254.77, 477.14] vs [367.33, 420.89]). As mentioned earlier, having less variance might be a good criterion to evaluate the preparation of the team. One of the options to evaluate the team success could be the ordering of the team standing by the highest values of their mean, median or minimum scores.

3. Summary

As stated at the beginning of the paper, this short report covers the initial phase of the “IOI data analysis” project. The current phase covers the analysis of the CMS database records for the IOI 2019 event and provides insights into the activities that happened in that year. More detailed insights will be covered in the next phase, which is going to cover the year on year analysis, a high-level analysis of the code that was submitted (how the code is being evolved, what is being added/changed on each successful attempts, and so on) and also have a software that shows the introduced graphs in real-time. This software will be visible to the Technical and Scientific teams to better understand what is happening behind the scenes and prevent the possible problems before they might happen.

Summarizing the current report, the following outcomes might be useful for event organizers, committees, and coaches:

- The teams are mentioned but never evaluated or rewarded. Team members that keep almost the same standing or steadily increase their overall position from year to year are indicators of the systematic approach (having fair and decent Nationals, trials, selection processes, preparation camps, participation in other contests, and so on).
- The organization processes shall be refined as much as possible. The data of the previous events and the previous contest day shall be analyzed to offer an efficient contestant seating plan, contestant desk design, the content of snacks, and other details. Using graphical icons and tags in contestant requests (like in IOI 2019 – #water, #banana, #question, etc) may help better categorize the requests.
- The dynamics of the task submissions in the previous years might be analyzed for efficient task type selection and the creation of the test cases.
- Coaches may analyze the behavior of the successful contestants and teams, to find patterns that lead to success and adjust their strategy accordingly.

The sources of the reports and results of the work can be found in a public GitHub repository of the project (GitHub, 2021b).

Acknowledgements

The authors would like to thank the IOI International Committee for their support of the "IOI Data Analysis" grant. This research is the 1st phase of the "IOI Data Analysis" project supported and sponsored by IOI grant announced in 2020.

The computational resources used in this research have been provided by ADA University's Center for Data Analytics Research (CeDAR).

References

Alnahhas, A., Mourtada, N. (2020). Predicting the performance of contestants in competitive programming using machine learning techniques. *Olympiads in Informatics*, 14, 3–20. DOI: 10.15388/ioi.2020.01.

CMS (n. d.). *Contest Management System*. `https://cms-dev.github.io`

Fantozzi, P., Laura, L. (2020). Recommending tasks in online judges using autoencoder neural networks. *Olympiads in Informatics*. 14, 61–76. DOI: 10.15388/ioi.2020.05.

GitHub (2021a). *ADA-SITE-JML / IOI-Grant: Structure of the CMS DB*. `https://github.com/ADA-SITE-JML/ioi-grant/tree/master/database`

GitHub (2021b). *ADA-SITE-JML / IOI-Grant: Project Page*. `https://github.com/ADA-SITE-JML/ioi-grant`

IOI Statistics (2020a). *International Olympiad in Informatics Statistics*. `https://stats.ioinformatics.org/olympiads/`

IOI Statistics (2020b). Rezults The ranking of IOI 2020 contestants. `https://stats.ioinformatics.obrg/results/2020`

Jovanov, M., Mihova, M., Kostadinov, B., Stankov, E. (2018). New approach for comparison of countries' achievements in science olympiads. *Olympiads in Informatics*, 12, 53–68. DOI: 10.15388/ioi.2018.05.

Kostadinov, B., Jovanov, M., Stankov, E. (2018). Platform for analysing and encouraging student activity on contest and e-learning systems. *Olympiads in Informatics*, 12, 85–98. DOI: 10.15388/ioi.2018.07.

Lodi, M. (2020). Informatical thinking. *Olympiads in Informatics*, 4, 113–132. DOI: 10.15388/ioi.2020.09.

Pankov, P.S., Kenzhaliyev, A.A. (2020). Pattern recognition and related topics of olympiad tasks. *Olympiads in Informatics*, 14, 143–150. DOI: 10.15388/ioi.2020.11.

Tsvetkova, M.S., Kiryukhin, V.M. (2020). Top 10 key skills in Olympiad in Informatics. *Olympiads in Informatics*, 14, 151–167. DOI: 10.15388/ioi.2020.12.

Vegt, W. van der, Schrijvers, E. (2019). Analyzing task difficulty in a Bebras contest using cuttle. *Olympiads in Informatics*, 13, 145–156. DOI: 10.15388/ioi.2019.09.

J. Hasanov is an Assistant Professor of Computer and Information Sciences in the School of IT and Engineering at ADA University. Dr. Hasanov is mainly focused on image processing and machine learning problems covering text and digital object recognition domains. Additional to the research field, Dr. Hasanov teaches the management aspects of the IT in production and operation environments. Dr. Hasanov has been an ITC member for the period of 2017–2020 and led HTC during IOI 2019.

H. Gadirli is a student in a dual-degree master program of "Master of Science in Computer Science and Data Analytics" at ADA University and George Washington University. Habil received his B.Sc. in computer science from ADA University in 2019.

A. Bagiyev is a first-year student in the dual-degree master's program of "Master of Science in Computer Science and Data Analytics" offered by ADA University and George Washington University. Aydin has been a member of the HTC team during IOI 2019 and contributed to the automation of the contestant environment management and image distribution processes. He also works as a full-time computer science instructor at ADA School, to contribute to the enhancement of the youths in the country.

Olympiads in Informatics, 2021 Vol. 15, 37–52
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.04

Security of Grading Systems

Martin MAREŠ
Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
e-mail: mares@kam.mff.cuni.cz

Abstract. Programming contests often employ automatic grading of solutions. Graders need to run potentially malicious code, which brings many security issues. We discuss various attacks on grading system security and suggest counter-measures.

Keywords: automatic grading, security, sandbox, covert channels.

1 Introduction

Many of the world's programming contests are based on *automatic grading* (evaluation) of programs submitted by the contestants. Basically, the grading process takes a submitted *solution,* compiles it, executes it on a set of *test inputs,* and checks if the produced outputs are correct. Execution of the program is usually subject to a *time and memory limit,* so that full score is awarded only to efficient solutions. A typical example of such contest is the International Olympiad in Informatics (IOI), but many others exist.

The *grader* usually forms a part of a larger *contest system,* which also handles distribution of task statements, collection of submissions, displaying of the rank list etc.

Not all contestants play fair. Some of them attempt to cheat in order to gain an unfair advantage. Cheating can involve seeking forbidden advice from outside sources, but also breaking integrity of the contest system. We will study the latter type of cheating.

We will focus on the grader, because this is the component which runs potentially malicious code provided by the contestant. We will study various security issues pertaining to graders. We will discuss various possible attacks on security of grading. We will propose measures to thwart the attacks, or at least to mitigate their effects.

We are not the first to study grading security. Early research on this topic is summarized in the survey by Forišek (2006) and in the paper by Sims (2012). However, the landscape of contest systems has greatly changed since that time.

Contest systems now routinely employ advanced sandboxes based on kernel namespaces and control groups – for example Isolate by Blackham and Mareš (2012). Peveler *et al.* (2019) propose using full-edged containers to encapsulate untrusted code. New security features are added to the underlying operating systems. New types of contest tasks and new programming languages are introduced. These ask us to extend the traditional view of what a solution is supposed to do – for example, runtimes of several languages need multiple threads of execution. All these news bring their security challenges.

Several security problems mentioned by Forišek are no longer relevant (we will discuss these in section 2). Some continue to be abused in even more creative ways (denial-of-service attacks described in section 3, covert channels via grader feedback in section 5.2). Most importantly, many new types of attacks have surfaced (sections 4 to 7).

Some of the attacks were never published, but circulated in oral communication in the rather small communities of contest organizers and contest system developers. We had the luck of working in the International Technical Committee of the IOI, where many of these communities intersect. This paper is our attempt at making the issues more widely known, so that they could be avoided in as many contests as possible.

Still, our ambitions fall short of covering all the contests in the world. We will focus on IOI-like contests and graders running on the Linux operating systems. Our findings probably apply to other contests to some extent.

2. Obsolete attacks

We will start our exposition with several types of "obsolete" attacks. These were quite popular in the past (and mentioned in Forišek's survey), but they are no longer possible in current contest systems.

2.1. *Timing*

Contest systems need to measure execution time in order to enforce time limits. This is usually done using CPU time accounting provided by the operating system. The Linux kernel keeps track of total run time of each process separately. In theory, whenever a process is scheduled to a CPU for some time slice, the duration of the slice is added to the run time of the process.

However, reading the system timer at every context switch was traditionally considered too slow. Because of that, statistical sampling was used instead: At every tick of the system clock, the tick was charged on the currently running process.

This is reasonably accurate if the time slices are long (relatively to the ticks), but much less accurate with rapid context switches. Most importantly, as the sampling is not random, it can yield misleading results if the behavior of the processes is correlated with the system clock.

Indeed, it is possible to construct a program which uses carefully planned pauses synchronized with the system clock to avoid being caught running. Whenever the timer tick interrupt arrives, the program is sleeping. This way, the measured run time will be close to zero, even though the program could have been running for the most of the time.

This technique was described by Tsafrir *et al.* (2007), but at that time it was already well known in the community of kernel developers.

In early 2007 (kernel 2.6.13), Linux switched to the Completely Fair Scheduler. Timer-based sampling was abandoned in favor of reading the hardware clock explicitly. The new scheduler makes the class of attacks described above ineffective. We refer the reader interested in details to a survey of Linux scheduling by Iskhov (2015).

Even with the new scheduler, run time measurements cannot be perfectly precise. Most importantly, parallel processes influence each other in various ways. Temporarily giving the CPU to another process often flushes the caches. Processes running concurrently on multiple cores compete for memory bandwidth and sometimes also for access to level-3 cache. These effects were thoroughly analyzed by Mareš (2011) and the conclusions are still valid.

2.2. *Time-of-check to Time-of-use Race Conditions*

Early contest systems sandboxed graded processes using the `ptrace` syscall.[1] This is a mechanism intended chiefly for use by debuggers. It allows to stop a program whenever it attempts to make a syscall. A supervising process (a debugger, or in our case a sandbox manager) can then inspect the arguments of the syscall and decide whether the syscall should be allowed. Depending on that, the process is either allowed to continue or aborted.

At the first sight, this approach seems to be straightforward: we can enumerate a short list of "safe" syscalls and forbid all the others. However, even very simple functions in standard system libraries use a surprising variety of syscalls internally. All these must be included in the safe list, so it is actually quite long and it is no longer easy to argue that all syscalls on the list are safe under all circumstances.

More importantly, this approach is effective on single processes, but once we have two processes or threads sharing memory, it becomes fundamentally insecure. Suppose that a thread A invokes the `open` system call to open a file. One of the arguments of `open` is a pointer to the file name, stored as a string in memory. The thread is stopped by `ptrace`, the sandbox manager checks that the file name is innocent (e.g., it is the input file of the task). So it decides to let the thread A continue. At that time, thread B wakes up and overwrites the memory by a different file name. So the actual execution of the `open` syscall accesses the other file.

[1] *Syscall* is a system call from the user space to the kernel. See Kerrisk *et al.* (2012) for a description of the syscall interface.

This is a typical case of a so-called TOCTOU (time-of-check to time-of-use) vulnerability. It is almost impossible to avoid in sandboxes based on filtering of system calls. So these sandboxes are suitable for single-threaded programs only.

Fortunately, current contest systems mostly switched to sandboxes based on kernel namespaces. Instead of limiting the set of available operations, they limit the reach of these operations. For example, it is permitted to create network sockets, but the socket can connect only to other sockets within the same sandbox. This approach was pioneered in contest systems by Mareš and Blackham (2012), see their paper for description of one such sandbox and further discussion.

Namespace-based sandboxes avoid the TOCTOU problem, because the kernel reads the parameters from the user memory only once.

Still, the discussion in this section stays relevant. Some container supervisors (e.g., `systemd-nspawn`) apply system call filtering as an additional security measure. This typically involves the `seccomp` mechanism in the kernel (see Edge (2015)). It uses either a simple bitmap of permitted syscalls numbers, or a program for the BPF virtual machine. Since the BPF program can access only direct arguments of the syscalls and not contents of user memory, the TOCTOU issue does not occur. The problems with syscalls required by the standard libraries still remain, though.

3. Denial-of-Service Attacks

The easiest type of attacks on most computer systems are so-called *denial-of-service (DoS) attacks*. The attacker tries to consume as much resources as possible, reducing availability of the system to other users. We will discuss various ways of mounting a DoS attack on a grading system.

During on-site contests, DoS attacks happen rarely and usually by mistake – there is nothing to be gained. They occur more frequently in Internet contents, and sometimes also in practice sessions of on-site contests where the contestants try to push the system to its limits.

3.1. *Execution Time*

Submitted programs can consume various resources during their execution. It is trivial to write an infinite loop, infinite recursion, or an infinite loop allocating memory. All these are easily stopped by time and memory limits enforced by the sandbox.

Less frequent cases involve so-called fork bombs – programs iterating the `fork` syscall to create new processes. This usually leads to exponential growth in the number of processes running. Since individual processes are small, having a per-process memory limit does not help. Instead, a global a memory limit on all processes together must be imposed (e.g., as done by Isolate via process control groups). Alternatively, we can limit the number of processes running inside the sandbox (e.g., by the `setrlimit` syscall).

Last, but not least, the program can fill up disk space by generating large files. Limiting file size using `setrlimit` is not enough, because multiple files can be created. A disk quota should be configured instead for the user ID under which the sandbox runs.

3.2. Compilation Time

When a contestant submits the source code of their solution, it has to be compiled first. Even though contest systems developers concentrate mainly on security of running the solution, many attacks can be performed in the compilation stage, too.

First of all, it is easy to write a short program whose compilation takes an arbitrarily long time. For example by chaining C++ templates:

```
#include <map>
using namespace std;

typedef map<int,int> M1;
typedef map<M1,M1> M2;
typedef map<M2,M2> M3;
// ...
typedef map<M15,M15> M16;

int main() { M16 tmp; return tmp.size(); }
```

This construction was already mentioned by Forišek. We confirm that it still works with current compilers (e.g., GCC 8.3.0), even though the template chain must be made longer to produce significant delay. (It should be no surprise as the C++ templates are known to be Turing complete, see for example Veldhuizen (2003).)

The compiler can be also trapped by `#include "/dev/zero"` – a special file containing an infinite sequence of zero bytes. GCC desparately looks for an end-of-line character, which would terminate the infinitely long line, and buffers the contents of the line in memory. This usually exceeds the memory limit sooner than the time limit.

It is also possible to include `/dev/urandom` instead – a source of infinitely many cryptographically strong pseudo-random numbers. The pseudo-random generator is sufficiently slow to make GCC exceed the time limit first.

Finally, the compiler can be also made to generate a very long output file. A particularly easy method is to create a huge static array with only the last element initialized (a completely uninitialized array would not be stored explicitly):

```
#define N 1000000000
char huge[N] = { [N-1] = 1 };
```

4. Attacks on In-Process Graders

At most programming contests, solutions read their input and write their output as files. The International Olympiad in Informatics uses a different interface since 2010: the task specifies an API between the solution and the *grader*.[2] The contestants have to write a program which conforms to this API – they implement functions called by the grader, and they can call other functions provided by the grader. The same approach is also used for interactive tasks at other contests.

Let us consider the following task as an example: The first player thinks of a secret integer S between 0 and 10^6. The goal of the second player (implemented by us) is to guess this integer using at most 20 queries of the type "Is the integer S less than x?".

The API consists of two functions:

- `int play()` – implemented by the contestant. Plays the game and returns the guessed integer.
- `bool less_than(int x)` – implemented by the grader. Returns if the secret number is less than the given x.

During compilation, the solution is linked with the grader. The grader contains the entry point of the whole program (the function `main()`). It generates the secret integer and calls `play()` defined in the solution. The `play()` calls `less_than(x)` back in the grader, which compares the x with the secret number stored in the grader and answers accordingly. At the end, `play()` returns to the grader, which compares the result with the secret number and stops the program. The exit code of the program can be used to signal whether the solution was correct.

In more complex tasks, there can be an input file read by the grader, and the grader can write the output to an output file, so that it can be checked by the other parts of the contest system.

The API-based interface has numerous advantages. It hides implementation details from the contestants. This can be important in languages, where efficient access to files requires various kinds of trickery (Java, but to some extent also streams in C++). In case of interactive tasks, it relieves the contestants of thinking about flushing output buffers properly. Furthermore, the low overhead of function calls (as opposed to inter-process communication) makes it easy to implement interactive tasks with lots of interaction.

Unfortunately, the security of this approach is very bad. We have a system with two components: a trusted grader and a completely non-trusted solution. We need to establish some kind of a security boundary between the components, so that the solution cannot influence the grader in unintended ways. A shared address space within a single process makes for a very poor security boundary. The solution can access all the data of the grader and it can even modify its code.

[2] Technically speaking, the term "grader" is misleading here. It is not the complete grading component of the contest system, but only a small piece capable of checking correctness in the particular task. However, we will stick to the term grader as used in IOI jargon for this chapter.

Graders used in past IOIs took effort to make the attacks less likely to succeed (and unintended crashes caused by bugs in solutions less likely to happen). They used cryptic hard-to-guess names for their internal functions and variables. They were writing a special secret string (a “cookie”) at the start of the output file, which makes it easy to detect that the output was not written properly by the grader. The cookie itself was stored in an obfuscated form inside the grader. Also, the input file was obfuscated and decoded by the grader.

All these measures certainly make attacks harder to succeed and easier to detect. Still, from the security point of view, they are typical examples of “security by obscurity”. And indeed, these were not completely effective in practice and some successful cheating attempts were discovered.

4.1. *Exchanging Library Functions*

Let us consider the typical case of a C++ solution, linked statically (as done by the CMS contest system at the IOI; the situation would be similar, though different in details, with dynamic linking). The linker combines four pieces of code: three object files (the solution, the grader, and a piece of startup code provided by the compiler) and the standard C library. The library is a collection of object files, usually one file per library function or a small group of functions.

The linker collects the initial object files and looks at their dependencies. They have the form of symbol references – an object file can ask for an arbitrary symbol (a named function or external variable). The linker tries to find the definition of this symbol in the other object files first. Only if it is not defined there, the linker inspects the library, finds an object file defining that particular symbol, and adds this object file to the current set of object files. The new object file can reference further symbols from the library and so on. The whole process is repeated until all symbol references are resolved.

Even though the grader uses cryptic names for all internal symbols, it still needs to call functions from the standard library (e.g., to read from files) and these have fixed names. But library functions can be easily interposed by the solution, since symbol definitions in other object files have priority over those in the library. For a concrete example, when the grader calls `write()` and there is `write()` defined in both the solution and the library, the grader will call the version from the solution.

This makes it possible for the solution to modify the output of the grader, which already contains the secret cookie. Of course, the solution must write the output using a different library function (e.g., `writev()`), or to invoke a syscall directly.

This attack can be mitigated by using two-step linking: first link only the grader with the standard library, but tell the linker to produce another object file as a result. In the second step, link this object file with the solution and again with the standard library. All calls to library functions in the grader are already resolved in the first linking, so the second linking cannot re-bind them to symbols from the solution.

4.2. *Rolling Back Grader State*

There is a much more serious attack on graders which keep some internal state to check correctness. In our toy example with guessing of numbers, the grader keeps a counter of queries, so that it can reject solutions asking more than 20 queries. The attacker could be interested in manipulating this counter to ask more queries with impunity.

Gaining enough information about internals of the grader to locate the counter in memory is hard to do under time pressure of the contest. Especially if the value of the counter is obfuscated or the grader stores multiple copies of the counter.

However, there is a much easier way: Simply make a copy of all data belonging to the grader and copy it back later. This allows to make a snapshot of the complete state of the grader and revert the grader to that state later.

Still, finding out what memory belongs to the grader is non-trivial. But the solution can easily identify its own memory, so it can copy all memory except its own. A good starting point is `/proc/self/maps`, which lists all virtual memory areas in the address space of the current process.

It is easy, but it can be made even easier if the solution is allowed to create multiple processes. The standard way of creating a new process is the `fork()` syscall – it creates a clone of the current process, which differs only by the process ID and the return value of `fork()`. So the solution can fork a new process, which has a copy of the grader's state. Then it can run a part of its computation inside that process, pass the results back to the main process, and exit. The main process still keeps the original state of the grader, not aware of any queries which were made in the new process. Since process creation is cheap, this can be easily iterated.

4.3. *Proper Design of Graders*

We described many security problems with in-process graders. In our opinion, attempts at fixing them are futile, because a proper security boundary between two machine-code programs inside the same process is impossible to maintain. We firmly believe that security by obscurity must be rejected and replaced by a mechanism that is secure by design.

In case of batch tasks (read input, then produce output), the grader can implement the task's API by reading and writing of files (or the standard input and output). All checking of correctness should be done outside the sandbox after the solution stops.

Interactive tasks are more subtle. If the amount of interaction is not too large, the grader can convert the API functions to communication over a pipe (or a UNIX-domain socket) with a separate grader process running outside the sandbox.

For tasks which require very intensive interaction, the latency of communication over a pipe can be prohibitively large. In such cases, a block of shared memory guarded by simple spinlocks can be used instead. The grader should copy the data to its local memory first to avoid TOCTOU issues mentioned in Section 2.2. This type of communication is more complicated and it needs further investigation. We are still convinced that security is worth the effort.

5. Covert Channels

In many contests, the solutions are tested on test data stored inside the grading system, but supposed to be unknown to the contestants. Under some circumstances, it is possible for the contestants to gain some information about the secret test data. This is a classic example of what security researchers call a *covert channel* between two components separated by a security barrier. We will describe several kinds of covert channels.

5.1. *Secrets Lying on the Disk*

Surprisingly often, secret input files, or even the correct outputs, could be found somewhere in the file system, readable to the solution. This was usually a result of bad configuration caused by human mistake. Namespace-based sandboxes make it much less likely to happen, since only an explicitly selected subset of the directory hierarchy is available inside the sandbox.

Execution environment of solutions is usually very restricted, but the compilation environment not much so. Compilers tend to require access to lots of unexpected files. It is therefore tempting to grant compilers read-only access to huge subtrees of the directory hierarchy. We will show that all information available to the compiler is effectively available to the solution, too.

Of course, if the compiler has access to a reference solution, the contestant's solution can simply use the C/C++ preprocessor to `#include` the reference solution. This however does not help to gain access to test data, because test data seldom conform to syntax of a C++ fragment, which could be included.

In such cases, file contents can be obtained using inline assembly. The GCC compiler does not produce machine code directly. It generates symbolic instructions and feeds them to an assembler (in case of GCC, it's the GNU Assembler, gas). GCC extends the C language by adding an `asm` statement, which can pass arbitrary strings directly to the assembler. This can be used to execute not only any machine instructions, but also arbitrary compiler directives (pseudoinstructions) of the assembler. One such directive is `.incbin`, which includes complete contents of a specified file byte by byte in the object file being produced.

Let us see a simple example, which embeds contents of `/etc/passwd` in the compiled program and prints it out when the program is run.

```
asm("\
   .data\n\
   start_hack: .incbin \"/etc/passwd\"\n\
   end_hack:\n\
    .text\n\
");

extern char start_hack[], end_hack[];

#include <unistd.h>
int main() {
    write(1, start_hack, end_hack - start_hack);
}
```

The `.data` directive switches to the data section, so that the contents of the file become part of the program's data. The `.text` directive switches back to the code section. The start and end of the file are demarcated by the `start_hack` and `end_hack` labels, which are accessible as external variables of array type in C code.

5.2. *Grader Feedback as a Covert Channel*

Some contests provide feedback on submissions, which allows contestants to fix errors and re-submit the solutions. All such feedback can be used as a covert channel, although of very small bandwidth.

Suppose that every submission is tested on 10 different test cases. The feedback for each test case includes the verdict (correct / wrong answer / time limit exceeded / runtime error), time used (in milliseconds) and memory used (in megabytes). Time limit is 5 seconds, memory limit 256 megabytes.

We estimate how much information about a single test case can be encoded in the feedback. We will ignore the verdict. We can generate an arbitrary memory usage between 1 and 256 megabytes, so we can encode 8 bits in that. Encoding information in execution time is more subtle as the value is influenced by measurement errors. According to Mareš (2011), random noise does not typically exceed small tens of milliseconds. So we will let the program run for a multiple of 100 ms and round the reported time to the nearest such multiple. This gives us 49 values between 0.1 s and 4.9 s, which can be used to encode 5 bits.

In this simple example, we can gain 13 bits of information about every test case per submission. The number of submissions is usually limited, so we cannot exfiltrate full test data from the contest system. But it is certainly possible to extract sizes of test inputs and other basic characteristics.

Because of this, contest organizers should put the amount of feedback under close scrutiny and try to minimize the effective bandwidth of the covert channel, while keeping the feedback useful to honest contestants.

Counter-measures can include providing only summary statistics like maximum execution time and memory over all test cases, and the histogram of verdicts (e.g., 3 times "passed", 5 times "wrong answer", and twice "time limit exceeded").

Furthermore, task designers should avoid tasks with very small entropy of inputs.

5.3. */proc File System*

Finally, we mention one unexpected source of potential information leaks: the `/proc` file system. It is a virtual file system whose files contain information about currently running processes (hence the name) and also global information on the state of the system (e.g., how much memory is used for what purpose). This is where commands like `ps` or `top` obtain their knowledge.

Although some information on processes (e.g., the set of descriptors of open files) is available only to the owner of the process, a small subset is available to all users. The public subset includes arguments given to the program when it is executed.

Visibility of the arguments is considered well-known among UNIX programmers, so people traditionally avoid passing passwords and other sensitive strings as arguments. However, we would like to remind the authors of contest systems and competition tasks about this issue. Process arguments should be never used for passing any information which should stay secret from the contestants. This includes things like test case number if the grader of a test case is started as a separate program. A better choice is to pass the information in environment variables, which are not considered public.

Sandboxes based on process namespaces mitigate this problem, because the `/proc` file system inside the sandbox reports only the processes living in the namespace of the sandbox. In addition to processes and threads of the solution, this includes only the sandbox manager itself. The Isolate sandbox makes an extra effort to hide even the sandbox manager, so that sandbox settings passed as arguments are not visible.

6. Cross-Language Attacks

New programming languages are growing at a surprising pace. Contest organizers would prefer to make available more languages than C++. A popular choice is Python with the intention of making the contest more accessible to beginners. A typical problem with Python is its speed, or lack thereof. IOI-level tasks typically require strict time limits, so that it is possible to distinguish between solutions of different time complexity. But a time limit which allows an optimal solution in C++ to run with a generous margin of 200 %, is grossly insufficient for an optimal solution in Python.

Organizers of competitions often try to sidestep this issue by allowing a higher time limit for Python programs. This necessarily involves a rather philosophical question of fairness, which we will avoid here. Instead, we will demonstrate that any such configuration can be easily misused to run a C++ solution with Python time limits.

The idea is to compile a C++ program, embed the resulting binary in a Python program, which will reconstruct the binary and run it.

Let us show a concrete example. We take a test program `test.cpp`, compile it to an executable file and strip off all debugging symbols to save space:

```
g++ -O2 -s test.cpp -o test
```

Then we use Python to compress the executable file and encode it in Base64 format. Base64 consists purely of printable ASCII characters, which can be embedded in our Python "solution".

```
import zlib, base64
bin = open('test', 'rb').read()    # Read the binary
compr = zlib.compress(bin, 9)      # Compress it
b64 = base64.b64encode(compr)      # Encode it in Base64
print(b64.decode('us-ascii'))      # Print the result
```

The solution itself will decode the executable file, write it to the current directory and execute it from there:

```
import base64, zlib, os
b64="""... output of the previous program ..."""
compr=base64.b64decode(b64)        # Decode Base64
bin=zlib.decompress(compr)         # Decompress

with open('hack', 'wb') as f:      # Write the binary to a file
   f.write(bin)

os.chmod('hack', 0o777)            # Grant executable permission
os.system('./hack')                # Execute
os.unlink('hack')                  # Cover up tracks :)
```

This form of the attack works in contests which pass input and output using files or standard input and output. In contests with an IOI-style API, it is necessary to call Python functions of the API from the C++ program. To accomplish this, the C++ program should be compiled as a Python extension and loaded from the Python program using `import`. Technically, Python extensions written in C++ are dynamically linked libraries with a well-defined interface to the Python interpreter. They can export functions callable from Python, call Python functions, and access state of the Python interpreter. We have a proof-of-concept implementation of such attack, but it is too large to include in this paper. It is still possible to write it within the few hours of a contest.

There are multiple potential mitigations. Generous limits on source code size can be lowered. The basic form of our attack requires writing to files, so the whole filesystem can be mounted read-only (and standard output connected to a writable file staying outside the sandbox). Or the writable directory can be mounted with the `noexec` option, so that code cannot be executed from it.

These mitigations are not ultimately effective. Writing to files can be replaced by creating a virtual file residing in memory using the `memfd_create` syscall, which is available in Python since version 3.8. It is still necessary to specify a file name when executing the program, but the name can refer to an open file descriptor using `/proc/self/fd/`N.

It is tempting to forbid tricks of this kind in contest rules, but we suspect that there are many variants on this theme and some of them could look innocent, or at least as corner cases of the rules. Similar methods can be used for most, if not all, other pairs of languages – see the `AnyExec2C` tool by Lukeš and Šraier (2020).

7. Other Attacks

In this section, we would like to mention two attacks, which do not fall into more general categories.

7.1. *Using Threads to Increase Cache Size*

Contests often allow starting of multiple processes or threads, because it is required by runtime environments of some languages (most notably Java and the .NET runtime). In such cases, time and memory limits are imposed on the group of processes as whole, typically using Linux control groups.

Initially, it seems that splitting computations to multiple threads does not yield any advantage. Even though the threads can run in parallel on a multiprocessor (or multi-core) machine, the time limit will be applied to the sum of the run times of the threads.

Surprisingly, there are cases where using multiple threads can improve run time. The reason is that current processors contain cache memory, which is much faster than the main memory, but much smaller. The effects of caching are quite complex, because processors often use several levels of caching. So we sketch just a simple estimate. The largest level of the hierarchy on current PCs (the L3 cache) holds several megabytes of data and it is shared among a subset of cores. Relatively to access to the L3 cache, access to main memory is about 5 times slower and access to other L3 caches about 3 times slower.

Suppose that the working set of our program (the data it accesses frequently) exceeds the size of a single L3 cache by a small factor. If the program runs on a single core, it will have only the L3 cache of that core available, so the data will not fit in the cache. If we split the program to multiple threads, they can run in parallel on multiple cores and have access to multiple L3 caches. Even though access to remote L3 cache is slower than to the local L3 cache, it is still substantially faster than access to main memory.

This attack is hard to carry out and quite simple to stop: bind the sandbox running the solution to a specific core. This way, even if the solution starts multiple threads, all of them will be scheduled in alternating timeslices on that single core. Obviously, the same computation can be performed by a single sequential program, so there is no possible gain from parallelism. (Except for possibly simplifying control ow of an interactive program, but that is hardly a cheat.)

7.2. *Storing Data in Socket Buffers*

If a task uses particularly tight memory limits, contestants can find numerous ways of storing data outside the address space of the their process. Depending on the method of measuring memory use, these ways will or will not be counted. If the sandbox uses traditional UNIX resource limits (see the `setrlimit` syscall), they will not be. If it uses memory control groups, they will.

A particularly nice example is the socket buffers – for each socket (an abstraction of a network connection), the kernel keeps a queue of data which was accepted for transmission, but not yet received by the other end of the connection. For TCP connections, the size of the queue is controlled by the tunable parameter `/proc/sys/net/ipv4/tcp_rmem` and the default maximum size is 6 MiB. By establishing 17 TCP connections to itself, a solution can keep 100 MiB of data in socket buffers.

7.3. *Security Issues in Processors*

Security issues are not limited to software. In January 2018, the world was caught by surprise by publication of several hardware vulnerabilities in processors. The most severe of these vulnerabilities are known under the names Meltdown (Lipp *et al.* (2018)) and Spectre (Kocher *et al.* (2019)). Meltdown affects only Intel CPUs, some forms of Spectre apply to all processors which employ branch prediction. Other similar problems were discovered later.

The underlying principle of all attacks of this class is the presence of subtle, but measurable side-effects of instructions which were executed only speculatively, but later rejected, because the speculative assumption turned out to be false. This can be used to create covert channels across various kinds of security barriers. For example, the original proof-of-concept code for Meltdown allowed an ordinary process to access kernel data. Spectre can be misused by JavaScript programs running inside the web browser to steal cookies belonging to other sites.

After a huge effort, these vulnerabilities were mostly mitigated in software, most importantly in the Linux kernel. The mitigations successfully prevent covert channels between different processes at the cost of slowing down all programs slightly. Only in cases where the security boundary goes within a single process, like in web browsers, additional per-process mitigations are necessary.

In our opinion, contest systems developers need not be worried about this class of attacks. Work-arounds implemented in the Linux kernel are sufficient in all cases where untrusted code is sandboxed properly.

8. Conclusion

We surveyed many possible attacks on security of contest systems. Some of them rather theoretical, some of them already observed in real contests. We discussed defenses against the attacks. We will summarize our recommendations here.

The contest system should be properly separated to trusted parts (e.g., the grader) and parts running untrusted code (execution of contestant's solution, but also its compilation). All untrusted code should be carefully sandboxed, there should be a narrow and well-defined interface between the sandbox and the trusted code. Every sandbox should place limits on total time and memory consumption, but also on total disk space used.

In particular, it is crucial to avoid any mixing of grader code responsible for checking correctness with any contestant's code in the same process. We outlined possible solutions in section 4.3. The case of interactive tasks with intensive interaction needs further investigation.

All feedback given to constants provides a potential covert channel for exfiltrating information on secret test data. The exact form of feedback should be chosen carefully to minimize the bandwidth of this channel.

Discriminating between solutions based on their programming language turns out to be inherently problematic. Section 6 suggests some mitigations, but they are necessarily incomplete.

Computer security is a rapidly developing discipline full of surprises (cf. the security issues of CPUs in section 7.3). As the saying goes, attacks never become weaker with time. We must continue our study, investigate new ways of attacking systems, and build our systems to be even more resilient.

References

Blackham, B., Mareš, M. (2012). A new contest sandbox. *Olympiads in Informatics*, 6, 100–109.

Edge, J. (2015). A seccomp overview. In: *Linux Weekly News*, 2015-09-02. Available on-line at: `https://lwn.net/Articles/656307/`

Forišek, M. (2006). Security of programming contest systems. In: V. Dagienė, and R. Mittermeir (eds.), *Information Technologies at School*, 553–563. Available online at `https://people.ksp.sk/~misof/publications/copy/2006attacks.pdf`

Ishkov, N. (2015). *A Complete Guide to Linux Process Scheduling*. M.Sc. thesis, School of Information Sciences, University of Tampere, Finland. Available online at `https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf`

Kerrisk, M. *et al.* (2012). *The Linux Man-Pages Project*. Available on-line at `http://www.kernel.org/doc/man-pages/`

Kocher, P. *et al.* (2019). Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*. Available on-line at `https://ieeexplore.ieee.org/iel7/8826229/8835208/08835233.pdf`

Lipp, M. *et al.* (2018). Meltdown: Reading kernel memory from user space. *27th USENIX Security Symposium (USENIX Security 18)*. Available on-line at `https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf`

Lukeš, S., Šraier, V. (2020). `AnyExec2C`. Available on-line at `https://github.com/exyi/anyexec2C`

Mareš , M. (2011). Fairness of time constraints. *Olympiads in Informatics*, 5, 92–102.
Peveler, M., Maicus, E., Cutler, B. (2019). Comparing jailed sandboxes vs containers within an autograding system. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*.
Sims, R.W. (2012). *Secure Execution of Student Code*. Technical report of Department of Computer Science, University of Maryland. `http://honors.cs.umd.edu/reports/simspaper.pdf`
Tsafrir, D., Etsion, Y., Feitelson, D.G. (2007). Secretly monopolizing the CPU without superuser privileges. In: *USENIX Security Symposium 2007*. pp. 239–256. Available on-line at `https://www.usenix.org/legacy/event/sec07/tech/full_papers/tsafrir/tsafrir_html/`
Veldhuizen, T.L. (2003). *C++ Templates are Turing Complete*. DOI: 10.1.1.14.3670. Available on-line at `https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670`

M. Mareš is a lector at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, organizer of several Czech programming contests, member of the IOI Technical Committee, and a Linux hacker.

Olympiads in Informatics, 2021 Vol. 15, 53–74
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.05

Why You Should Know and Not Only Use Sorting Algorithms: Some Beautiful Problems

László NIKHÁZY, Áron NOSZÁLY, Bence DEÁK
Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
e-mail: nikhazy@inf.elte.hu, noszalyaron4@gmail.com, deakbence2002@gmail.com

Abstract. In most programming languages, the built-in (standard library) sort() function is the most convenient and efficient way for ordering data. Many software engineers have forgotten (or never knew) the underlying algorithms. In programming contests, almost all of the tasks involving sorting can be solved with only knowing how to use the sort() function. The question might arise in young students: do we need to know how it works if we only need to use it? Also, why should we know multiple efficient sorting algorithms, is not one enough? In this paper, we help the teachers to give the best answers to these questions: some beautiful tasks where the key to the solution lies in knowing a particular sorting algorithm. In some cases, the sorting algorithms are applied as a surprisingly nice idea, for example, in an interactive task or a geometry question.

Keywords: programming contests, sorting algorithms, competition tasks, geometric algorithms, teaching algorithms.

1. Introduction

Today's students are best motivated to learn by seeing exactly how they can use their knowledge in the future. A computer programmer frequently uses library functions implemented by other programmers, particularly often the built-in or standard library data structures and algorithms of programming languages (e.g., the C++ standard library). We can define increasing levels of knowledge related to them: the interface, the complexity of operations, the theoretical background, and the implementation details. The question naturally arises: how much should students know about them? It depends largely on the level of students and the concrete algorithm or data structure itself.

We focus on talented high school students whose aim is to perform well at competitions. Without question, they need to know the interface and the complexity of standard library elements to succeed in competitive programming. Also, most of them will learn the theoretical backgrounds during university studies in computer science. Is it useful to teach them ahead of university? Well, it makes sense to examine this question separately for different algorithms and data structures. In this article, we focus on the sorting algorithms. We would like to make a point that students should know several efficient and less efficient sorting algorithms if they want to excel at national competitions and participate in international ones by showing some problems where this knowledge is a great advantage. With a short survey presented in section 2, we verified that it is a matter worth investigating.

Section 3 presents six tasks from various sources that can be solved by modifying a sorting algorithm, but the standard library sort() function is not applicable. These problems might be given as follow-up tasks after teaching the corresponding sorting algorithm to demonstrate some unusual applications. In multiple cases, the topic of the task is from another domain, e.g., geometry, interactive problems, or graph theory, and the fact that a field of computer science helps tackle a problem in another area makes them beautiful solutions, in our opinion. When discussing implementation details of the solutions, we consider the features of the C++ standard library since it is the most used language in competitive programming (Halim *et al.*, 2013).

2. A Quick Survey

We conducted a little, non-representative survey with 26 students participating from the most talented young Hungarian programmers aged between 13 and 19 years. The questions were about Quicksort (Hoare, 1961a), Merge Sort (Knuth, 1997c), Insertion Sort (Knuth, 1997b), and Minimum/Maximum Selection Sort (Knuth, 1997d, referred to as Selection Sort in the following), whether they know these algorithms, and whether they

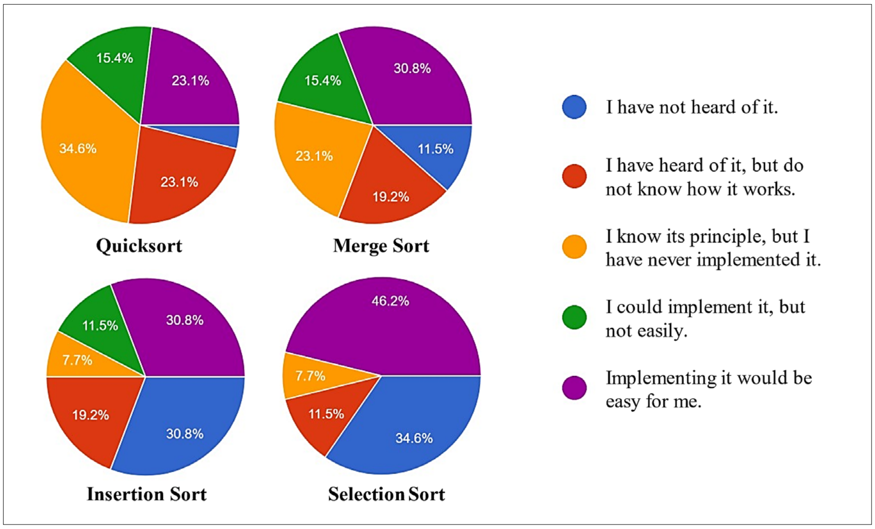


Fig. 1. Results of a non-representative quick survey with gifted pupils about sorting algorithms.

have implemented them already. We also asked if they use the built-in sort() function often, and more than 95% of the students have used it already, 80% very frequently. The results presented in Fig. 1 are intriguing and justify the importance of the topic of this article.

Surprisingly, the two simpler and less efficient sorting algorithms are less known, although students are, in general, more confident in implementing them. Still, it is true for all four algorithms that more than half of the students have never implemented them, and typically about 30% of the pupils think that the implementation would not be a problem for them. We want to mention that we also included a verification question about each algorithm's worst-case complexity, and out of those who stated that they know it, about 30% replied wrongly for Quicksort, 10% for Merge Sort, and 7% for Insertion Sort.

3. Tasks and Solutions

3.1. *Matching Nuts and Bolts*

This problem was first mentioned as an exercise in (Rawlins, 1992, p. 293). Since then, there have been numerous publications related to fast deterministic solutions (Alon *et al.*, 1994; Bradford, 1995; Komlós *et al.*, 1998). We present a version that was featured in the Hungarian IOI Qualification Competition, 2019.

Statement (shortened)

There are N bolts of distinct sizes in one box and N corresponding nuts in another one. Your task is to find the matching nut for every bolt. However, the only operation you can perform is comparing the size of a bolt and a nut by trying to match them, and you have a limited number of such queries. You cannot compare two bolts or two nuts.

This is an interactive task. Inside the solution, you can call the check(i, j) function at most M times in total, to compare the i^{th} bolt to the j^{th} nut, and it will return -1, 0, or 1 if the nut is smaller, the same, or bigger than the bolt, respectively. Important note: the solution is fixed throughout the interaction, i. e. the grader is not adaptive.

Subtasks and constraints

In the Hungarian contest, there was a single subtask: $M = N^2/4$, and $N \geq 40$. However, there is a bit more than a single idea in this task, so there could be other subtasks with different limits, to award suboptimal and simpler solutions as well. We suggest some:

i) $N = 2, M = 2$
ii) $N = 10, M = 100$
iii) $N = 100, M = 5000$
iv) $N = 1000, M = 250\,000$
v) $N = 10^5, M = 10^7$

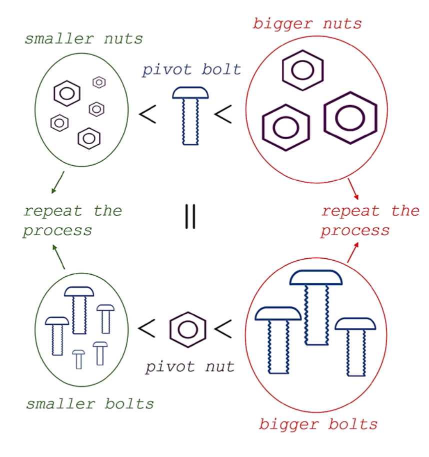
Solutions

We can select the pair of a bolt by trying out all the N nuts, leading to an $M = N^2$ solution, which is enough to solve subtask ii. If we exclude the nuts for which the pairs have already been found, we can solve the task with $M = N\,(N-1)/2$ operations that is sufficient for subtasks i–iii. How can we improve this? We might use the information obtained when comparing a bolt with all the nuts – we can form two sets, the bigger and the smaller nuts. And here comes the interesting extra step: since we also have the matching nut, we can compare that with all the bolts to partition them into two sets, too. Those two sets of nuts and bolts will correspond to each other, so we can repeat the process within each of them. Fig. 2 highlights the main steps of the algorithm.

It is a classic divide-and-conquer solution very similar to the Quicksort algorithm (Hoare, 1961a). In fact, it is best to implement the above-shown partitioning by reordering the elements, just like in Quicksort. The number of comparisons is $O(N^2)$ in the worst case, when one of the partitions is always empty. Thus, it is crucial to introduce some randomization, otherwise, there might be a test where the worst case happens. It can be proven similarly to the complexity analysis of Quicksort (Cormen *et al.*, 2009, pp. 170–190), that with randomization, the average number of comparisons is $O(N \log N)$, which easily fits into the limits of subtasks iv–v, so we can expect that the solution passes for all testcases, maybe with tuning the random seed a couple of times.

Analysis

The algorithm is almost the same as Quicksort, but in this problem, two collections need to be sorted in parallel. This fact brings the sole difference: for partitioning one collection, we use an element of the other collection and repeat this step correspondingly in the reverse direction. Naturally, knowing the Quicksort algorithm means a massive advantage for solving this task. The connection is also very important for analyzing the



1. Randomly choose a *pivot bolt* and compare it to all the nuts, finding its pair (the *pivot nut*) and partitioning the nuts to two sets, the *bigger* and the *smaller nuts*.

2. Compare the *pivot nut* to all the rest of the bolts to form the sets of the *bigger bolts* and *smaller bolts*.

3. Repeat the above steps recursively with the set of *bigger bolts and nuts*, as well as the *smaller bolts and nuts* if their size is at least two.

Fig. 2. Steps of the Quicksort-like solution of the Matching Nuts and Bolts problem.

complexity of the solution and knowing that hitting the $O(N^2)$ worst-case complexity can be avoided by randomly choosing the pivot element. It must be noted that the non-adaptive nature of the grader is an essential detail in this regard. With this, we can rely on the knowledge that the expected complexity of Quicksort is $O(N\log N)$.

In general, the task can be seen as a sorting problem instead of the matching problem. It is sufficient to sort both the collections of nuts and bolts. Certain other sorting algorithms might be used as well. For example, we can do a Bubble Sort (Knuth, 1997a) by comparing a nut and a bolt at the same position in the sequence and swapping the bigger one (or any if they are equal) with the next element and then moving to the next position. After two such passes, the biggest items will be at the end of both sequences. Although this is an excellent idea, the complexity of this solution is $O(N^2)$.

Further questions

The solution presented above includes randomization. It is a natural question whether there is a deterministic algorithm with worst-case complexity better than $O(N^2)$ for this problem. There are numerous articles presenting different approaches. First, Alon and colleagues (1994) gave a deterministic $O(N\log^4 N)$ algorithm with the idea finding a good pivot for the partitioning in the Quicksort using expander graphs, later Bradford (1995) showed the existence of an optimal, $O(N\log N)$ algorithm, and finally Komlós and colleagues (1998) found an optimal solution based on the AKS sorting algorithm.

3.2. *Median Strength*

This task was included in the problem set of IOI 2000. Afterwards, Horváth and Verhoeff (2002) published an article about its detailed analysis. A variant of this problem appeared recently in the Qualification Round of Google Code Jam (2021).

Statement (shortened)

There are N objects arranged in a line, each having a unique strength and a label assigned to them. You can compare three (x, y, z) objects by calling a function $\mathrm{Med3}(x, y, z)$ which returns the label of the object with median strength. Your task is to write a program that, among all N objects, determines the one with median strength with no more than M calls to Med3. N is guaranteed to be odd.

Subtasks and constraints

In IOI 2000, there were no subtasks, the points were given for each test case, and they had various sizes. The upper limits were $N \leq 1499, M \leq 7777$.

Solutions

There are several different approaches to this problem, and (almost) all of them require some knowledge of sorting algorithms. The first is based on minimum/maximum selection, and Horváth and Verhoeff call it onion peeling. In each iteration, we eliminate the two extremes in the following way. First, we pick any two available objects. Then, we iterate through all the other ones, keeping track of the minimum and the maximum we processed so far. See the pseudocode for a better understanding (Fig. 3). After $(N-1)/2$ iterations (and $(N-1)^2/4$ calls to Med3), the only remaining element will be the median of the original set of objects.

The second approach is based on Insertion Sort. We start with obtaining a sorted list of three objects by a single call to Med3. (Note that we do not care if the order is reversed since the final median can be identified anyway.) Each following object z can be inserted in between some (x, y) pair of consecutive objects, increasing the length of the sorted prefix by one. By calling Med3(x, y, z) for each consecutive pair, we can make the following observation: the function returns y for some (possibly 0) initial (x, y) pairs, then z for at most one pair, and finally x for the rest of the pairs. (Med3$(x, y, z) = z$ for exactly one pair, unless object z is the new minimum or maximum of the sorted prefix.)

We may naively search for the place of insertion by trying all (x, y) pairs one by one. Although this linear search is $O(1)$ in the best case, its worst and average-case complexity is still $O(N)$. However, our observation above suggests that the suitable position can be found using ternary search with worst-case time complexity of $O(\log N)$. See the pseudocode below for the details of the ternary search (Fig. 4). (We need an additional call to Med3 beforehand to handle the case where z is a new extremum.)

As a third approach, Quicksort can also be modified according to our needs: we simply have to use a ternary partitioning algorithm. This involves choosing two pivot elements and dividing the array into three subarrays. Moreover, since we are only interested in finding the median, we can base our solution around Quickselect (Hoare, 1961b) instead of Quicksort. The difference is that we do not need to recurse into such subarrays,

```
while objects.size() > 2:
    x = objects[1]
    y = objects[2]
    for i = 3 to objects.size():
        med = Med3(x, y, i)
        if med == x:
            x = i
        if med == y:
            y = i
    objects.eraseAt(x)
    objects.eraseAt(y)
```

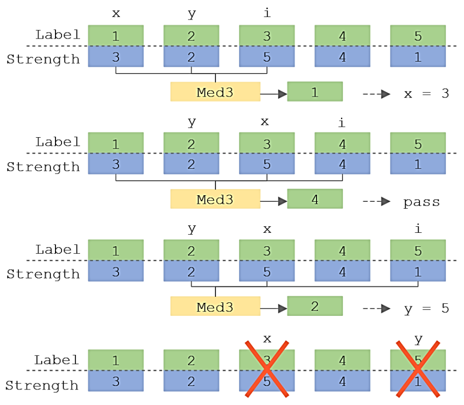


Fig. 3. Pseudocode and illustration of an iteration of the onion peeling method.

```
function ternary_search(l, r, z):
    if r == l + 1:
        return l
    d = (r - l) div 3
    x = l + d
    y = r - d
    med = Med3(x, y, z)
    if med == y:
        return ternary_search(y, r, z)
    elif med == z:
        return ternary_search(x, y, z)
    else:
        return ternary_search(l, x, z)
```

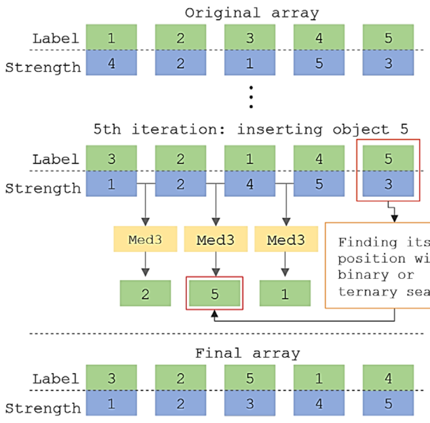


Fig. 4. Pseudocode of ternary search and overview of the insertion based method.

which are guaranteed not to contain the median. Although this improved algorithm still makes $O(N^2)$ calls to Med3 in worst case, the average time complexity is $O(N)$, which is sufficient to solve the problem.

Several interesting heap-based solutions have also been described in (Horváth & Verhoeff, 2002). These algorithms perform pretty well in practice, despite having an average/worst-case time complexity of $O(N \log N)$. We must note that there exist worst-case linear selection algorithms, given that we can compare any pair of elements in $O(1)$. Some of them, e.g. the Median of Medians (Blum *et al.*, 1973), could be modified to fit our case, but they are not advised to use due to their relatively high constant factor.

Further questions

This problem appeared with a different output in Google Code Jam 2021. In that variant, we must find one of the orders (non-descending or non-ascending) explicitly, so the application of some sorting algorithm is a straightforward idea.

We propose the following interesting, related problem. In a query, the median of any subset of objects can be asked, and the task is to determine the object with the K^{th} smallest strength. (The median of an even-sized set is the smaller one of the two middle elements.) Is it possible to describe an algorithm that uses at most $O(N)$ queries (if so, what is the upper bound)? What if we are not allowed to ask the median of two objects? The solution would be a special application of the Quickselect algorithm, in which we have the power to choose the median of all elements as the pivot. The worst-case linear complexity allows for giving a nice, $c \cdot N$ upper limit for the number of queries.

3.3. *Tournament*

This problem is about the algorithmic aspects of a famous theorem in graph theory, namely that every tournament (complete directed graph) contains a Hamiltonian path (Rédei, 1934). Hell and Rosenfeld (1983) describe an optimal algorithm similar to Binary Insertion Sort; we elaborate on applications of other sorting algorithms below.

The 2019 Nemes Tihamér Programming Competition (a Hungarian national programming contest for 9–10th grade students) included this task, phrased as constructing such an order of participants of a round-robin tournament, in which every player defeated the next in the sequence. We give an alternative problem statement that is less associated to sorting, which we heard from Lajos Pósa in his mathematics camps.

Statement (short versioN)

A faraway country consists of N islands. There is a strange public transportation system, dragons carry people between any two islands, but they only fly in one of the two directions. As a tourist, you want to take a route consisting of the most islands possible, without visiting any island twice. You can choose any island to start from.

This is an interactive task. The dir(i, j) function returns true if the dragons fly from the ith to the jth island, and false if they fly in the reverse direction. You can call this function at most M times. You should give the solution as a list of island numbers in the order of visiting. Attention: the grader might be adaptive, meaning that the return value of the dir(i,j) function may depend on the previous calls made to it.

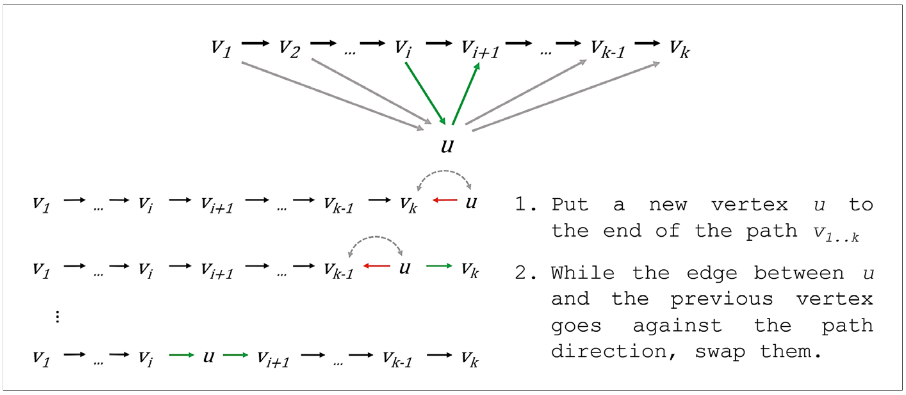
Subtasks and constraints

Making the problem interactive allows us to differentiate solutions based on the number of edges that they inspect. In the Hungarian competition, the complete graph was given as the input, which prevents distinguishing $O(N\log N)$ and $O(N^2)$ solutions. With M as the upper limit of queried edges, we suggest the following subtasks:

i) $N = 4, M = 5$
ii) $N = 10, M = 50$
iii) $N = 1000, M = 500\,000$
iv) $N = 100\,000, M = 2\,000\,000$

Solutions

In the following, $x \longrightarrow y$ denotes that the edge between x and y is directed towards y (i. e. dir(x, y) is true). Subtask ii can be solved with brute force; first, query all edges of the graph and then check all permutations of vertices. In subtask i, the number of queries is less than the number of edges ($5 < 6$)), so we need something smarter. If we query the 3 edges between 3 vertices, we can make a path of them: $x \longrightarrow y \longrightarrow z$. Then it makes sense to ask dir(y, w), where w is the fourth vertex. One can see that depending on the answer, it suffices to check either dir(x, w) or dir(z, w) to be able to insert w into the path.


$v_1 \to v_2 \to \dots \to v_i \to v_{i+1} \to \dots \to v_{k-1} \to v_k$
u
$v_1 \to \dots \to v_i \to v_{i+1} \to \dots \to v_{k-1} \to v_k \leftarrow u$
$v_1 \to \dots \to v_i \to v_{i+1} \to \dots \to v_{k-1} \leftarrow u \to v_k$
⋮
$v_1 \to \dots \to v_i \to u \to v_{i+1} \to \dots \to v_{k-1} \to v_k$

```
1. Put a new vertex u to
   the end of the path v1..k
2. While the edge between u
   and the previous vertex
   goes against the path
   direction, swap them.
```

Fig. 5. Solution algorithm of the Tournament problem based on Insertion Sort.

Is there always a path through all vertices, like in the case of $N = 2, 3, 4$? The answer is yes, and we prove it by giving an algorithm that finds this path. Consider a path consisting of k points: $v_1 \longrightarrow v_2 \longrightarrow \dots \longrightarrow v_k$. If there is a vertex u not included in this path, we can insert it as follows. If $u \longrightarrow v_1$ or $v_k \longrightarrow u$ then we can put u to one end of the sequence. Otherwise, looking at the edges between u and $v_{1..k}$, the first is an in-edge ($v_1 \longrightarrow u$) the last one is an out-edge ($u \longrightarrow v_k$), and we can find $v_i \longrightarrow u \longrightarrow v_{i+1}$ for some index i, since there must be a direction change at least once in the middle (see Fig. 5). Thus, the path can be extended to $v_1 \longrightarrow v_2 \longrightarrow \dots \longrightarrow v_i \longrightarrow u \longrightarrow v_{i+1} \longrightarrow \dots \longrightarrow v_k$. One possible implementation of this "repeated insertion" method goes as follows:

This is essentially the Insertion Sort algorithm if we treat $\mathrm{dir}(i, j)$ as the comparator function. It performs $N(N-1)/2$ comparisons in the worst case and solves subtask iii, but not iv. However, the number of comparisons can be reduced by using binary search for finding the position to insert: we maintain a range (v_l, v_r) within the path, where $v_l \longrightarrow u$ and $u \longrightarrow v_r$ (initially $l = 1$ and $r = k$), and it can be halved by checking the middle element. This method, which is analogous to Binary Insertion Sorting, is the algorithm described in (Hell & Rosenfeld, 1983), and it is easy to see that we check $O(N\log N)$ edges of the graph. On the other hand, for an efficient implementation, a data structure is needed that supports fast insertion and retrieval at any position, which is not available in the C++ standard library. Fortunately, we can come up with other solutions.

Viewing the task as a sorting problem brings new ideas. Let us examine Quicksort, for instance. Interestingly, it works without modification, as demonstrated in Fig. 6. After partitioning, there is $v_i \longrightarrow u$ for every v_i vertex on the left side of the pivot point u, and $u \longrightarrow w_i$ for every w_i vertex on the right. If we have a $v_1 \longrightarrow v_2 \longrightarrow \dots \longrightarrow v_l$ path on the left side and a $w_1 \longrightarrow w_2 \longrightarrow \dots \longrightarrow w_r$ path on the right, there will be a $v_1 \longrightarrow \dots \longrightarrow v_l \longrightarrow u \longrightarrow w_1 \longrightarrow \dots \longrightarrow w_r$ path containing all points. Since Quicksort does $O(N\log N)$ comparisons on average, it could also pass the big tests if the grader was not adaptive, but the $O(N^2)$ worst-case complexity makes it unsuitable with an adaptive grader.

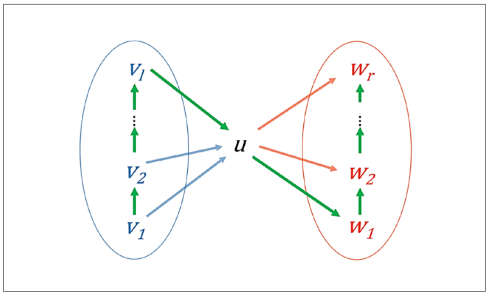


Fig. 6. The mechanism of Quicksort in the Tournament problem.

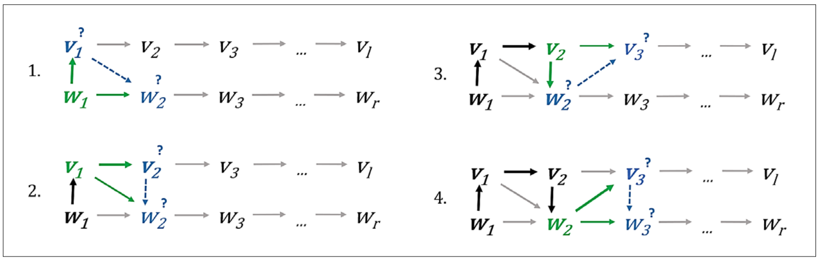


Fig. 7. The first few steps of merging two paths in the Tournament problem.

To decide whether Merge Sort is applicable, we should inspect the merging process of two paths. Fig. 7 shows that at any point when we add a vertex to the merged path, there are out-edges to both possible following points (the next element of the same sequence and the first element of the other). It follows that Merge Sort also solves the problem without any modification. Its worst-case $O(N\log N)$ complexity, and straightforward implementation makes it the best choice for the solution. What more, knowing that the std::stable_sort() in C++ standard library uses Merge Sort, it comes down to calling that on the sequence $1, 2, \ldots, N$, supplying dir(i,j) as the comparator. We must note here that the std::stable_sort() is not applicable in the same manner, because it uses Introsort (the GCC implementatioN), which is a hybrid of Quicksort, Heapsort, and Insertion Sort, and while Quicksort and Insertion Sort both produce a solution, Heapsort does not work for this problem, unfortunately.

Analysis

Finding the Hamiltonian path in a complete directed graph can be viewed as determining a generalized sorted sequence given an almost arbitrary relation. The generalized sorted sequence means that we only check the relation between consecutive elements ($x_1 \longrightarrow x_2 \longrightarrow \ldots \longrightarrow x_N$). The relation is arbitrary in the sense that we can choose $i \longrightarrow j$ or $j \longrightarrow i$ for any (i, j) pair of elements independently, but it still has some important properties:

1. Irreflexive, since there are no loop-edges in the graph.
2. Antisymmetric, because there cannot be both $i \longrightarrow j$ and $j \longrightarrow i$.
3. Semiconnex, since for all $i \neq j$ pairs $i \longrightarrow j$ or $j \longrightarrow i$.

Loop-edges are irrelevant to this problem, so instead of the irreflexivity, we might as well assume reflexivity, and then the semiconnexity becomes connexity (where $i = j$ is also included). Looking at the proofs of the three solutions above (Insertion Sort, Quicksort, Merge Sort), one can notice that they only rely on the connexity property of the relation. This fact is fascinating, namely that the mentioned sorting algorithms produce a generalized sorted sequence if the comparison relation is connex.

Methods that involve extremum selection, such as Selection Sort and Heapsort, normally require the relation to be transitive. Still, Wu (2000) managed to modify the Heapsort algorithm to solve this problem as well.

It is worth mentioning that the reasonings we gave above about the correctness of solutions can be made into precise mathematical proofs by complete induction. All of them would start with the assumption that every tournament graph of less than N vertices contains a Hamiltonian path. When we take a graph with N vertices, we divide it into smaller parts (maybe extracting just one vertex); by our assumption, there exist Hamiltonian paths within the parts, and we can transform them into one path. It is intriguing to see that sorting algorithms produce mathematical proofs in graph theory.

Further questions

A related more difficult problem was introduced in the 2016 Polish Olympiad in Informatics with the title Turysta (Szkopuł, 2016) that goes as follows. Given a complete directed graph, construct the longest path starting from each vertex. One can see that the strongly connected components form a complete directed acyclic graph and thus have a Hamiltonian path. Within a strongly connected component, there is a Hamiltonian cycle (Camion, 1959) that can be constructed similarly to the Hamiltonian path in a tournament. Thereafter, giving the longest path from any vertex is straightforward.

3.4. *Polyline with Acute Angles*

This beautiful geometry problem was featured in Codeforces Round #698 with the title Nezzar and Nice Beatmap (Codeforces, 2021a). It is also contained in the Timus Online Judge, called Mammoth Hunt (Ipatov, 2007), which states that it was introduced in the Ural State University Junior Contest, 2007. A paper by Fekete and Woeginger (1997) discusses the question more generally and includes one of the presented solutions.

Statement (shortened)

There are N points with integer coordinates given in the Cartesian plane. Connect all the points with a polyline that has only acute angles. More precisely, if we take any three consecutive points A, B, and C, the $\measuredangle ABC$ angle is strictly less than 90°. The 0° angle is also accepted (it is not to be confused with the 180°).

Subtasks and constraints

There are no subtasks in any of the task's two sources due to the contest format. The upper limit for the number of points is 5000, which suggests an $O(N^2)$ solution. In an IOI-style contest, it would make sense to introduce two more subtasks for small N, and there could be various other conditions for the set of points.

We propose some:

i) $N \leq 3$
ii) $N \leq 20$
iii) $x_i = 0$ or $y_i = 0$, $i = 1..N$
iv) The points form a regular N-sided polygon
v) $N \leq 5000$

Solutions

The first subtask is just about calculating the angles between three points. We use the $\measuredangle ABC$ notation for the angle between the AB and BC lines. The acuteness can be determined by taking the sign of the dot product of the BA and BC vectors: $\mathrm{dot}(BA, BC) = (A.x - B.x) \cdot (C.x - B.x) + (A.y - B.y) \cdot (C.y - B.y)$. If it is positive (meaning that the cosine of the enclosed angle is positive), then $\measuredangle ABC < 90°$.

The second subtask can be solved by backtracking. In subtask iii, we can visit the points of one axis by going in alternating directions and do an appropriate switch between the two axes. In subtask iv, a good idea is to always go to the opposite or next-to opposite vertex of the polygon. There are two beautiful $O(N^2)$ solutions to the general case.

In the first solution, we build the polyline one by one. The set of points are denoted by $V = \{P_1, P_2, ..., P_N\}$. We can start in an arbitrary $S \in V$ point and let us choose the most distant point T among the rest of the points, i. e. for which $|ST| \geq |SP_i|$ $(\forall P_i \in V)$. If there are multiple such points, we can choose any. It can be proven by contradiction that for any next P_i point $\measuredangle STP_i < 90°$. Indeed, as shown in Fig. 8, if $STP_i \geq 90°$ then $|SP_i| > |ST|$ because opposite the biggest angle is the longest side

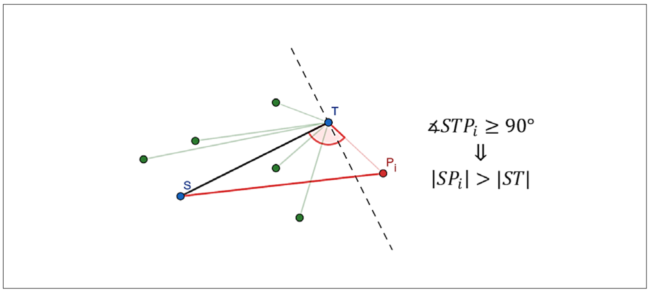


Fig. 8. The correctness of the most distant point method.

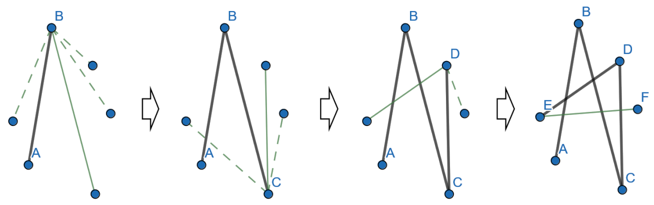
of a triangle (also true for degenerate triangles), and it is impossible since we chose T as the furthest point.

We can repeat this step, now take all the remaining points and choose the next point Q so that $|TQ|$ is maximal, i. e. $|TQ| \geq |TP_i|$ $(\forall P_i \in V \setminus \{ST\})$. The acuteness of $\measuredangle STQ$ is ensured by the choice of T as described above, and the condition for the Q point guarantees that the next angle is less than 90°, too, by the same reasoning. We keep doing the same until there are no points left. So, the algorithm goes as follows: see Fig. 9.

The complexity is clearly $O(N^2)$ if the maximum selection is done with a simple loop over all the remaining points. It is an interesting question whether it could be sped up using some data structure; we do not discuss it here.

The second approach is based on the following observation: any three points form a triangle (possible degenerate) where at most one of the three angles is not acute. This way, if there are three points A, B, C in the sequence such that $\measuredangle ABC \geq 90°$, changing their order to A, C, B or C, A, B will surely fix their angle. Let us inspect this algorithm: see Fig. 10.

For correctness, we need to prove that the three new angles introduced by the insertion of a P point are acute. See Fig. 11. (If one of the angles does not exist, the corresponding part can be skipped.) Assume that P is inserted between A_i and A_{i+1} in the acute polyline $A_1, A_2, \dots A_k$. Then $\measuredangle A_{i-1}A_iP < 90°$, because otherwise A_i and P would have been swapped. The other two angles, $\measuredangle A_iPA_{i+1}$ and $\measuredangle PA_{i+1}A_{i+2}$ must be acute because we observed another obtuse/right angle of the same triangle when P was swapped with one of the points previously. The complexity of this method is $O(N^2)$ as well, because when inserting a new point to a polyline of length k, we perform at most k angle calculations and swaps.



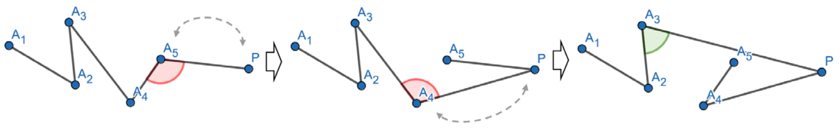
```
1. Choose an arbitrary point and add it to the polyline.
2. Repeat N-1 times: from the points not yet in the polyline select
   the one that is the furthest away from the lastly added point and
   add it to the polyline.
```

Implementation: reorder the elements

Idea: always choose the furthest point

```
for i = 1 to N - 2:
    maxi = i + 1
    for j = i + 2 to N:
        if distance(P[i], P[j]) > distance(P[i], P[maxi]):
            maxi = j
    swap(P[i + 1], P[maxi])
```

Fig. 9. The algorithm of repeatedly choosing the most distant point.



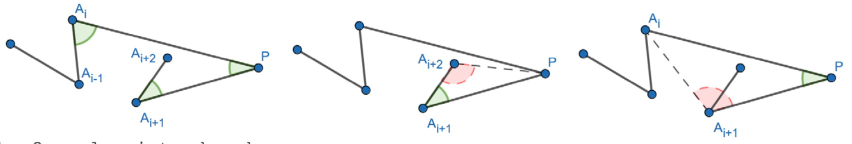
```
Repeat N times:
   1. Put a new point  P at the end of the polyline.
   2. While there is at least two points before  P and the angle in-between
      is not acute, swap  P with the previous one (as shown above).
```

Implementation: reorder the elements

Idea: we can find a position for inserting the new point

```
for i = 3 to N:
    j = i
    while j > 2 and not acute(P[j - 2], P[j - 1], P[j]):
        swap(P[j-1], P[j])
        j = j - 1
```

Fig. 10. The algorithm of the insertion method.



The 3 angles introduced by the insertion of P

(1) $\measuredangle A_{i-1}A_iP < 90°$ because the insertion process stopped there

(2) $\measuredangle PA_{i+1}A_{i+2} < 90°$ because $\measuredangle A_{i+1}A_{i+2}P \geq 90°$

(3) $\measuredangle A_iPA_{i+1} < 90°$ because $\measuredangle A_iA_{i+1}P \geq 90°$

Fig. 11. Correctness of the insertion method.

Analysis

The task can be viewed as an unusual sorting problem where every three consecutive elements must satisfy a relation, let it be denoted by $acute(A, B, C)$. The two general solutions presented above strongly resemble the Selection Sort and the Insertion Sort algorithms. The connection is especially clear in the case of the insertion method, seeing that we get a correct implementation if we simply replace the comparison in the Insertion Sort, as pointed out by the editorial in (Codeforces, 2021b). The method of repeatedly selecting the most distant point can be seen as a Selection Sort. Here the maximum is not something global; it is relative to another element. Namely, if we have a point S, we are looking for a suitable next point T, for which $acute(S, T, P_i)$ $\forall P_i \in V$. This is exactly what we proved when T is chosen as the furthest point from S.

Not all sorting algorithms can be used to solve the problem because of the strange nature of the $acute$ relation. The following property of the acute relation enables using Insertion Sort: if not $acute(A, B, C)$ then $acute(A, C, B)$ and $acute(C, A, B)$. This can be interpreted as the connexity property of this relation, and in the analysis of the previous problem (Tournament), we saw that Insertion Sort works for connex binary relations. Although the same is true for Quicksort and Merge Sort, they are unfortunately not applicable because the behavior of ternary relations differs significantly from binary ones.

Further questions

It remains a challenging open question whether there is an algorithm with better than $O(N^2)$ complexity to solve this problem. (Fekete & Woeginger, 1997) also poses this question, with no answer. We have the impression that there is a faster solution; however, we have not managed to find such. The task can be formulated in space or even more than 3 dimensions. The angle defined by 3 points in a d-dimensional space is calculated from the coordinates similarly as shown above. In fact, the exact solutions can be applied to solve the problem in multiple dimensions.

3.5. *Binary Manipulations*

This problem was introduced in the 4 th stage of the Ukrainian Olympiad in Informatics in 2019.

Statement (shortened)

We have a list of N numbers (N is a power of 2), the i^{th} element of this list is t_i. We want to sort this sequence with a limited amount of $move$ operations. In one $move$ operation, we can move a number from the position 2^x to the beginning of the list ($2^x \leq N$).

Subtasks and constraints

In the original contest, N was at most 128, and at most 16384 $move$ operations were allowed. There were also six subtasks:

i) $N = 2$, all numbers are different
ii) $N = 4$, all numbers are different
iii) $N = 8$, all numbers are different
iv) The list consists of $N/2$ zeros and $N/2$ ones in arbitrary order
v) All numbers are different
vi) No further constraints

Solutions

In the following, $move(i)$ denotes the move operation performed from the position i (where i is always a power of 2). The first subtask can be solved by a simple if statement. Subtasks ii–iii can be solved by breadth-first search as we have at most $8! = 40320$ different lists, and there are 3 transitions from each list. Subtask iv suggests a divide and conquer approach. Thus, we will assume that we can solve the problem for $K = 2^x$ and try to solve it for $2K = 2^{x+1}$.

First, let us sort the first K numbers. Then, if we perform K times a $move(2K)$ operation, we effectively moved the second part to the beginning, so we can sort that part, too. Now we have two sorted sequences which we want to merge into one. Let us indicate if a number should be in the first/second half of the sorted list by giving it a zero/one label. Obviously, after sorting the two halves, both the first and the second half starts with some number of zeros followed by some number of ones. Then we perform $move(K)$ until the ones in the first half form a prefix of the list. Afterwards, we perform $move(2K)$ operations while the last element is a one, which leads us to all ones in the first half and all zeros in the second. Now we have all the numbers in the wrong half, and they might be in the wrong order too. The first issue is easily fixed, and their order can be fixed, too, by sorting them again. So, in the last steps of the algorithm for sorting the $2K$ numbers, we sort the first K numbers, then perform a $move(2K)$ operation K times and sort the first K numbers again. The overall number of move operations in this algorithm is $T(N) = O(N^2)$ according to the Master theorem (Cormen *et al.*, 2009), since we have the recurrence $T(N) = 4T(N/2) + O(N)$. Overall, the biggest steps of the algorithm are the following: see Fig. 12.

In the middle of the algorithm, we have assigned zeros and ones to numbers based on which half a number should belong, but in the case when not all numbers are different, this is not well defined. To solve this issue, we can replace t_i by $t_i \cdot n + i$, so all numbers will be different while preserving their relative order.

Though we have described a solution that would fit in the limits of the problem, it is not asymptotically optimal as there is a method with $O(N \log N)$ $move$ operations on average. If the input is sufficiently random, then there is an $O(N)$ $move$ operation algorithm to partition along the median element at each recursive step. It is the fol-

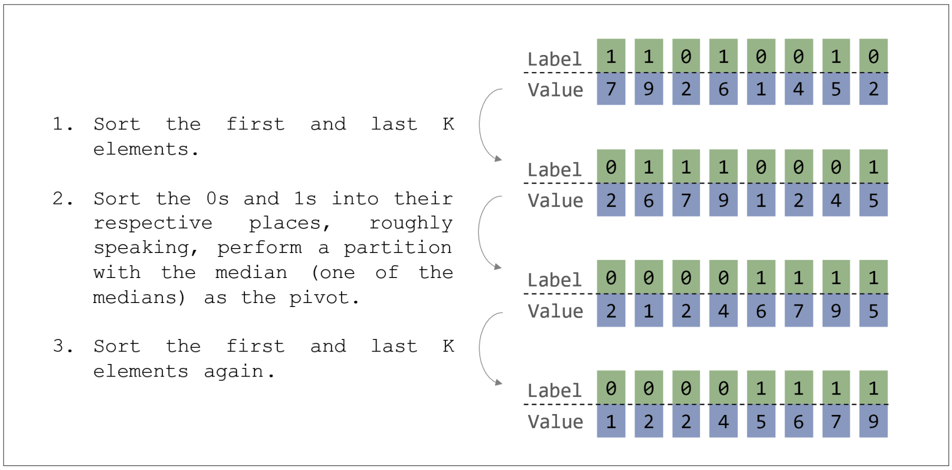

1. Sort the first and last K elements.
2. Sort the 0s and 1s into their respective places, roughly speaking, perform a partition with the median (one of the medians) as the pivot.
3. Sort the first and last K elements again.
Label 1 1 0 1 0 0 1 0
Value 7 9 2 6 1 4 5 2
Label 0 1 1 1 0 0 0 1
Value 2 6 7 9 1 2 4 5
Label 0 0 0 0 1 1 1 1
Value 2 1 2 4 6 7 9 5
Label 0 0 0 0 1 1 1 1
Value 1 2 2 4 5 6 7 9

Fig. 12. Overview of the solution of the Binary Manipulations problem.

lowing algorithm: while there is a one among the first K elements, if the K^{th} element is a one, perform a $move(2K)$, otherwise perform a $move(K)$. This method splits the zeros and ones into their correct half. This partitioning method has a worst-case complexity of $O(N^2)$ though if the input is not random enough. Note that it always terminates since, after each operation, the number of ones does not decrease in the second half.

Analysis

Both solutions presented here are divide and conquer algorithms, resembling Merge Sort and Quicksort. The first one has the properties of Merge Sort, as in each step we sort the halves recursively, but it is slower than usual since the merging step cannot be done without interfering with the already sorted subparts. This means that two more recursive calls are needed. It is quite a natural way to approach this problem as the powers of two suggest a divide and conquer approach, and the most well-known such sorting algorithm is Merge Sort.

The second described solution can be viewed as a Quicksort with a fixed pivot as the current subarray's median. This also arises naturally from knowing Quicksort as we do not have any effective way to select a random pivot due to the limited range of operations. The method used to make all numbers different is also an excellent lesson about the stability of sorting algorithms, as we have indirectly made our sorting algorithms stable with it, as well as making them work for different numbers.

Lastly, we measured the average number of move operations performed by the two algorithms presented. The Table 1 shows the average number of performed move operations (rounded to 5 digits) for different values of N, running the algorithms on 100 uniformly randomly generated tests. The results demonstrate clearly that the Merge Sort-like solution (MS) is expected to do much more move operations than the Quicksort-like approach (QS) on random inputs.

Further questions

Similar questions have been examined throughout the literature, for example, considering the "reverse" of the current $move$ operation, i. e. moving from the beginning to an arbitrary position (Aigner & West, 1987).

An easy follow-up question to this task might be: what if N^{th} is not a power of 2? Let us generalize a bit; we define some set S which contains the positions from which

Table 1

Average number of move operations done by the two solutions

N	16	32	64	128	256	512	1024
MS	169.94	682.59	2640.6	10121	38480	147790	571880
QS	95.28	283.79	781.58	2079.5	5270.2	13108	32031

a valid $move$ operation could be made. This means that in the original problem $S = \{2^i \mid 1 \le 2^i \le N\}$, whereas now we are trying to sort with $S = \{2^i \mid 1 \le 2^i \le N\} \cup \{N\}$. Questions might arise about what conditions are necessary and sufficient to sort with an arbitrary set S. The minimality can also be further explored as in regular swap-based sorting (Humphreys & Liu, 1996). For example, if $S = \{1..N\}$, we have a similar solution as in (Aigner & West, 1987). Designing an algorithm that works well for given S and input sequence might be interesting as well.

3.6. *Inversion Count*

We present an elementary problem, namely counting the inversions in a sequence, and its well-known solution using Merge Sort, also described in (Halim *et al.*, 2013, p. 355). We include it because it is a fundamental example of our topic.

Statement (shortened)

A pair of indices (i, j) is called an inversion of an array A if $i < j$ and $A[i] > A[j]$. Given an array of N integers, you should count the total number of inversions in it. For example, the array $[3, 1, 4, 1, 5, 9, 2]$ has 7 inversions: $(1, 2)$, $(1, 4)$, $(3, 4)$, $(1, 7)$, $(3, 7)$, $(5, 7)$, and $(6, 7)$.

Subtasks and constraints

i) $N \le 10^3$
ii) $N \le 10^5$

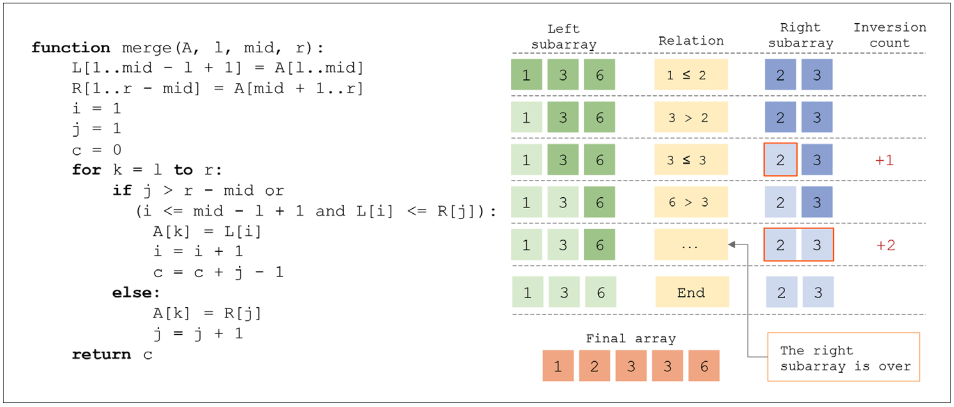
Solutions

A naive solution works for the first subtask: we simply iterate over all pairs (i, j) (where $i < j$) and increment the total count if $A[i] > A[j]$. The time complexity is $O(N^2)$ thus infeasible for the second subtask.

To solve the second subtask, we will use a divide and conquer approach based on Merge Sort; namely, with a slight modification of the merging step, one can obtain an answer in $O(N \log N)$ time.

Suppose we have to find the number of inversions in some subarray $[l; r]$ while also sorting it in non-descending order. The idea is to define a function $\text{count}(l, r)$ that solves the problem recursively. If $l = r$, then the answer is trivially 0, and the subarray is already sorted. In the following, we assume that $l < r$ holds. Let us denote the middle index as mid (formally, $mid = \lfloor (l + r)/2 \rfloor$). We can divide the problem into the following three subproblems:

1. The number of (i, j) inversions where $l \le i, j \le mid$.
2. The number of (i, j) inversions where $mid < i, j \le r$.
3. The number of (i, j) inversions where $l \le i \le mid$ and $mid < j \le r$.



```
function merge(A, l, mid, r):
    L[1..mid - l + 1] = A[l..mid]
    R[1..r - mid] = A[mid + 1..r]
    i = 1
    j = 1
    c = 0
    for k = l to r:
        if j > r - mid or
          (i <= mid - l + 1 and L[i] <= R[j]):
            A[k] = L[i]
            i = i + 1
            c = c + j - 1
        else:
            A[k] = R[j]
            j = j + 1
    return c
```

Fig. 13. Pseudocode and example of the modified merging step.

Subproblems 1 and 2 can be solved recursively by calling count(l, mid) and count$(mid + 1, r)$. Given that subarrays $[l; mid]$ and $[mid + 1; r]$ are sorted, the number of inversions of the 3rd kind can also be computed by applying a small change to the merging step of Merge Sort. While merging the two subarrays, we sum up the number of inversions using the contribution technique (Debowski, 2018). In other words, for all $l \leq i \leq mid$, we calculate the "contribution" of i via finding the amount of $mid < j \leq r$ indices such that (i, j) forms an inversion.

It is also worth noting that the total number of inversions is the same as the number of Bubble Sort-like swaps (consecutive swaps) needed to sort the array in non-descending order. This follows from the very fact that Bubble Sort gets rid of the inversions one by one (each consecutive swap accounts for exactly one inversion). So, another $O(N^2)$ solution would be to count the swaps during a Bubble Sort.

As a second note, we need to mention that there are different approaches to solve this problem optimally, many involving the use of some special data structure (e.g., order statistic tree, binary indexed tree, or segment tree).

Further questions

A natural question is whether we can use other asymptotically optimal sorting algorithms (e.g., quicksort, heap sort) to count the number of inversions. The answer is yes. For example, there are some less common, stable versions of Quicksort suitable for this problem.

4. Conclusions

We demonstrated various problems where the solutions are connected to sorting algorithms, in most cases unexpectedly so. In these tasks, the sort() function of the standard library cannot be used, the implementation of a particular algorithm is required. (There was one exception, the std::stable_sort() from the C++ standard library can be used to solve the Tournament task – but knowing the theoretical background is necessary even in this case.) All the presented tasks were involved in some competition for high school students, and while the frequency of such questions might be very low, we can still conclude that students need to know various comparison-based sorting algorithms if they want to be successful in national and international competitions. Hence, the problems above are beneficial for educational purposes.

In the Matching Nuts and Bolts task, the solution is a slightly modified Quicksort, while Bubble Sort could also work as a slower solution. The Median Strength problem has connections to many sorting algorithms: Selection Sort, Insertion Sort, Quicksort, and Heapsort. The Tournament problem is an example where Insertion Sort, Quicksort, and Merge Sort can be applied without modification. The Polyline with Acute Angles has a very surprising solution using the Insertion Sort with a ternary relation, while a connection to Selection Sort can also be made. The Binary Manipulations task is a neat restricted sorting problem where the knowledge of Quicksort and Merge Sort comes handy. Inversion Count has a well-known enlightening solution using Merge Sort along with a solid connection to Bubble Sort.

The Tournament problem and the Polyline with Acute Angles are examples where sorting algorithms are helpful to prove mathematical theorems. This reminds us of elegant proofs in which theories of another field of mathematics are applied, like Euler's proof of the existence of infinite primes using infinite series, or the fundamental theorem of algebra proved using complex analysis or even geometry, the probabilistic method used in combinatorics, and statements in Euclidean geometry proved by linear algebra or projective geometry, just to mention a couple of them.

By taking a close look at these few problems, we can observe a pattern of how Quicksort, Merge Sort, and Insertion Sort (also with binary search) can be helpful in many different scenarios, and Heapsort, Selection Sort and Bubble Sort could also be involved in some cases. Moreover, although our aim was to point out some problems where they appear directly, we believe that the biggest gain of learning these sorting algorithms is the capability of applying their principles in other situations.

It would be worthwhile to examine the question similarly for other elements of the standard library. For instance, the priority queue (heap), lower and upper bound (binary search), set and map (dynamically balanced binary search trees), unordered set and map (hash tables), among others. We think that the answer is a clear yes for binary search since it is also a basic optimization strategy, much less clear for dynamically balanced binary trees, considering that we have not seen a need to implement them, although there is a curious application of the C++ set with a custom comparator to find a pair of intersecting segments (CP-Algorithms, 2018), where the internals of the data structure need to be known to a certain extent. The question is definitely interesting for the heap and hash tables; if there were similar problems, they would be profitable for education.

References

Aigner, M., West, D.B. (1987). Sorting by insertion of leading elements. *Journal of Combinatorial Theory, Series A*, 45(2), 306–309.

Alon, N., Blum, M., Fiat, A., Kannan, S., Naor, M., Ostrovsky, R. (1994). Matching nuts and bolts. SODA,

Blum, M., Floyd, R.W., Pratt, V.R., Rivest, R.L., Tarjan, R.E. (1973). Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4), 448–461.

Bradford, P.G. (1995). Matching nuts and bolts optimally.

Camion, P. (1959). Chemins et circuits hamiltoniens des graphes complets. *Comptes Rendus de l'Académie des Sciences de Paris*, 249, 2151–2152.

Codeforces. (2021a). *1477C. Nezzar and Nice Beatmap*. Retrieved March 31 from `https://codeforces.com/contest/1477/problem/C`

Codeforces. (2021b). *Editorial of Codeforces Round #698*. Retrieved March 31 from `https://codeforces.com/blog/entry/87294`

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to algorithms*. MIT press.

CP-Algorithms. (2018). *Search for a pair of intersecting segments*. Retrieved March 31 from `https://cp-algorithms.com/geometry/intersecting_segments.html`

Debowski, K. (2018). *Sums and Expected Value*. Codeforces Blog. Retrieved March 31 from `https://codeforces.com/blog/entry/62690`

Fekete, S.P., Woeginger, G.J. (1997). Angle-restricted tours in the plane. *Computational Geometry*, 8(4), 195–218.

Google. (2021). *Median Sort – Analysis*. Google Code Jam. Retrieved March 31 from `https://codingcompetitions.withgoogle.com/codejam/round/000000000043580a/00000000006d1284#analysis`

Halim, S., Halim, F., Skiena, S.S., Revilla, M.A. (2013). *Competitive programming 3*. Citeseer.

Hell, P., Rosenfeld, M. (1983). The complexity of finding generalized paths in tournaments. *Journal of Algorithms*, 4(4), 303–309.

Hoare, C.A.R. (1961a). Algorithm 64: quicksort. *Communications of the ACM*, *4*(7), 321.

Hoare, C.A.R. (1961b). Algorithm 65: find. *Communications of the ACM*, *4*(7), 321–322.

Horváth, G., & Verhoeff, T. (2002). Finding the median under IOI conditions. *Informatics in Education*, 1(1), 73–92.

Humphreys, J.F., Liu, Q. (1996). *A course in group theory* (Vol. 6). Oxford University Press on Demand.

Ipatov, A. (2007). *1578. Mammoth Hunt*. Timus Online Judge. Retrieved March 31 from `https://acm.timus.ru/problem.aspx?space=1&num=1578`

Knuth, D. (1997a). Sorting by Exchanging. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 106–110). Addison–Wesley.

Knuth, D. (1997b). Sorting by Insertion. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 80–105). Addison–Wesley.

Knuth, D. (1997c). Sorting by Merging. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 158–168). Addison–Wesley.

Knuth, D. (1997d). Sorting by Selection. In *The Art of Computer Programming* (3rd ed., Vol. 3: Sorting and Searching, pp. 138–141). Addison–Wesley.

Komlós, J., Ma, Y., Szemerédi, E. (1998). Matching nuts and bolts in O (n log N) time. *SIAM Journal on Discrete Mathematics*, 11(3), 347–372.

Rawlins, G.J. (1992). *Compared to what?: an introduction to the analysis of algorithms*. Computer Science Press New York.

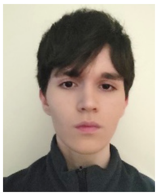
Rédei, L. (1934). Ein kombinatorischer Satz. *Acta Litteraria Szeged*, 7, 39–43.

Szkopuł. (2016). *Turysta*. Polish Olympiad in Informatics. Retrieved March 31 from `https://szkopul.edu.pl/problemset/problem/kqBM3UKWL-qlFiXIOxPXL35m/site/?key=statement`

Wu, J. (2000). On finding a hamiltonian path in a tournament using semi-heap. *Parallel Processing Letters*, 10(04), 279–294.

L. Nikházy is a Ph.D. student at Department of Media & Educational Informatics, Faculty of Informatics, Eötvös Loránd University in Hungary. His current research interest is computer programming talent education. He started his career as a software engineer at Google, but he gradually shifted to computer science education, and now devotes all his time to teaching talented children. He has been actively involved in organizing programming competitions and team coaching in Hungary in the past 4 years.

Á. Noszály is a Computer Science BSc student at the Faculty of Informatics, Eötvös Loránd University in Hungary. He has participated in several international programming competitions including the IOI. His current interests are in mathematics, problem setting and grading system development.



B. Deák is a first-year BSc student of Computer Science at the Faculty of Informatics, Eötvös Loránd University. He has been participating regularly in mathematics and informatics competitions since his early high school years, and now takes part in preparing several national programming contests in Hungary. He is interested in most subfields of theoretical computer science and discrete mathematics.

Olympiads in Informatics, 2021, Vol. 15, 75–82
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.06

Automatic Makers as a Source for Olympiad Tasks

Pavel S. PANKOV, Taalaibek M. IMANALIEV,
Azret A. KENZHALIEV
Institute of Mathematics, Kyrgyzstan
American University in Central Asia, Kyrgyzstan
Korea Advanced Institute of Science and Technology (KAIST)
e-mail: pps5050@mail.ru, imanaliev_t@auca.kg, azret.kenzhaliev@gmail.com

Abstract. Automatic maker is meant as a device to produce (or compose) things by itself, by a model, customization or a program. Nowadays the most advanced automatic makers in mass use are 3D-printers.The purpose of the paper is to present methods of generating various Olympiad tasks by using evident images of virtual automatic makers. As a perspective, production (performing) of processes is also considered (as automatic 4D-makers). Besides of the main operation: putting a pixel (voxel), the following primitives with virtual things can be involved: cutting; gluing; putting a building block; erasing a building block; copying a fragment, (for 4D-) shifting a building block. Tasks are generated naturally: to make a given thing (perform a given simple process) optimally in any sense (with respect to time; to number of primitives; to number of building blocks etc.). Such tasks are well-understood, have short formulations and are difficult to be solved even with initial data of small volume; a "brute force" method is either inapplicable or gives too overestimate of complexity.

Keywords: Olympiad, informatics, automatic maker, optimization tasks, 3D-printer.

Introduction

Automatic maker is meant as a device to produce (or compose) things by itself, by a model, customization or a program. Since ancient times, various copying and duplicating equipment may be considered as predecessors of such makers. Since computers appeared, they could produce 2D-things by means of plotters and (2D-)printers. Nowadays the most advanced in mass use are 3D-printers. Industrial robots can produce almost all of these.

Remark. It is interesting that "3D-printer" is called so only by analogy. Its principle is quite different.

The purpose of the paper is to present methods of generating various Olympiad tasks by using evident images of virtual automatic makers.

As a perspective and as a source for tasks, we consider production (performing) of processes also. Such devices may be considered as 3D(space)+1D(time)=4D-makers, we also consider 1D(line)+1D(time)-makers; 2D(plane)+1D(time)-makers.

Considered primitives: besides of the main operation: putting a pixel (voxel), cutting ("by a lazer beam"), putting a building block; erasing a building block; copying a fragment, (for 4D-) shifting etc.

Simplified schemes of such makers generate Olympiad tasks naturally: to make a given thing (perform a given simple process) optimally in any sense (with respect to time; to number of primitives; to number of building blocks etc.).

For solving an Olympiad task by the contestant, in addition to the common limits in the CPU time (traditionally 1 second) and in memory (traditionally 256 megabyte) there exists an actual limit (*) on average time to write and debug a program even if the contestant has necessary skills and vision of the algorithm (about 1–1.5 hours). Sometimes such limit is achieved by means of a long-winded and complicated text of the task, with many "permissions" and "bans" (**). In proposed tasks this limit is achieved naturally, because of their geometrical content.

We hope that proposed tasks on automatic makers can be done well-understood, to have "short and elegant formulations" (Dagienė *et al.*, 2007) and are difficult to be solved even with initial data of small volume; a "brute force" method is either inapplicable or gives too overestimate of complexity.

On the other hand, such tasks are "natural" in sense of (Pankov, 2008): a human can "see" the answer for corresponding task image with initial data of small volume without calculations.

Methods to generate and examples of such tasks are considered in the paper.

We will write "input" in examples in abbreviated form: lines are separated with the sign \ .

Three examples:

Task 1. Spear. (Pankov *et al.*, 2018)

It is known that Japan appeared as Drops into Ocean from Spear (see the picture). Let us formalize and optimize this process to create an archipelago. Divide Ocean into equal squares. Let each Drop be a square of Land.

Task: given a $K \times K$-binary matrix ('0's mean Ocean, '1's do Land) and the set of possible steps of Spear. Initially Spear is over the North-East corner of the matrix.

How many steps of Spear are necessary to create all Lands (to pass all '1's)?

The simplest sufficient set of possible steps of Spear is {S, W, E}.

Example for K=11: Input: 11. \

```
00000000000000
00000000000100
00000000001110
00000000000000
00000000011000
00000000110000
00000011110000
```

```
00011111000000
00000000000000
00010110000000
00010000000000
```

Output: 32.

[Possible beginnings of the optimal ways: WWSES… or SWWSE…]

Task 2. 3D-printer (Kyrgyzstan Regional Olympiads, February 2020)

The *X*- and *Y*-axes are horizontal, the *Z*-axis is down. Sides of all cubes are equal 1 and parallel to the axes, coordinates of their centers are integer numbers. The lower semi-space "$Z \le 1$" is filled with cubes. The initial coordinates of Cube-printer are (0, 0, 0). Given one, two or three cubes with coordinates in $[-N, N] \times [-N, N] \times [1, N]$. At each step Cube-printer moves by one along one of axes and erases a met cube. How many steps of Cube-printer are necessary to erase the given cube(s) (with cube(s) over them only)?

Example for two cubes: Input: 2 \ 8 7 1 \ 9 7 5. Output: 21.

It is seen that the complexity of this task does not depend on $N > 10$.

Task 3. Robot (Kyrgyzstan Regional Olympiads, March 2021, improved). Cubic Robot of volume 1 moves in continuous media (for instance, the warm iron cube moves in dense snow). Its edges are parallel to *X-, Y-, Z*-axes. A "shift" of Robot is its motion along or against one of the axes by an integer number J ($|J| \leq 10^{12}$). Given a sequence of 2..6 shifts, find the volume of Robot's "trace" (empty space in media made by Robot's motions including Robot itself).

Example for three shifts. Input: 3 \ Z 5 \ X –6 \ X 4. Output: 12.

We present examples of tasks but we do not propose any general algorithms to solve such tasks. On the contrary, we suppose that such algorithms do not exist and preferences of such tasks are that each task demands its own algorithm, with little discoveries, to smooth out the effect of training contestants.

1. Media and Operations

We will use the following spatial primitives: segments of length 1 on 1D; squares in 2D (pixels) and cubes in 3D (voxels), their sides are equal 1 and are parallel to the axes; coordinates of their centers are integer numbers. We will name all them "spexels".

We will consider 3D-printer in general sense: arranging arbitrary set of spexels in space. It is convenient to do it in zero gravity otherwise certain supports are necessary. We will mention "transparent spexels" for this purpose. (For generation of tasks, "transparent pixels" will be used too).

Temporal step is equal 1.

To present processes, discretizing the idea of Eulerian coordinates in continuum mechanics we propose time-space primitives: space primitives existing during one temporal step. Their indices are space coordinates of their centers and temporal one (natural number). We will name them "timexels".

An automatic time-maker works as follows. It arranges spexels and waits one temporal step. Then it (very quickly, by the idea of cinema) re-arranges (in any way) next spexels for the next step.

There can be homogeneous and non-homogeneous spexels and timexels, for instance, of different colors.

Remark. By our opinion, the terms "spexels" and "timexels" can be applied to other objects which are in use, for instance, regular triangles and regular hexagons, triangles in splines.

We will use the following operations:

Usual *putting* of spexel or timexel to the given spot (with given coordinates).

Cutting along sides (endpoints) of spexels ("by a laser beam"). There are two kinds of cutting in 2D and 3D cases: cutting along all straight line or plane; cutting along one common side of two spexels (1-*cutting*).

By custom, adjacent spexels are suggested to be glued. Hence, the operation of *gluing* can be applied either to previously cut spexels or to a "new" one.

Erasing a spexel.

Moving a spexel. We will consider moving along the coordinate axes.

Moving some glued spexels.

Repainting spexels.

2. Tasks on Spatial Arrangements

Task 1 – Solution. If permitted steps are within "a cone" ({S, W, E} or {S, SW, SE, W, E}) then dynamic programming can be applied, the complexity is $O(K^3)$.

Otherwise, for instance for {S, N, W, E}, complexity increases to $O(L^2 * 2^L)$, where L is the number of '1's in the matrix. A dynamic programming on bitmasks where we enumerate all '1's and compute the states *dp[mask][latest_visited_bit]* can be applied – *mask* denotes which '1's were visited by the spear and *latest_visited_bit* denotes which of those '1's marked by the mask was visited last. We iterate over possible next unvisited '1's ("off" bits in mask) and update transition states.

`https://ideone.com/cMkaMZ`

Task 2 – Solution. The limit (*) is vast: there are many different cases. To pass the following tests a "discovery" of zigzag path is to be made:

Test 2.1. Input: N=2; C1=(0,0,20); C2=(0,1,10). Output: 30.

Test 2.2. Input: N=3; C1=(0,0,30); C2=(0,1,20); C3=(2,0,10). Output 60.

Task 3 – Solution. If the limit for length of shifts $|J|$ is not "large" ($|J| \leq 10^6$) then creation of array of all cubes in Robot's trace solves the task

`https://ideone.com/app8e4`

For "large" $|J|$ we offer the following algorithm.

Let the initial position of Robot be (0 0 0).

Define a "segment" by six integer numbers. For instance, the first shift in Example (Z 5) generates the segment X = 0, Y = 0, Z = 0..5 or (0 0 0 0 0 5). The second shift (X -6) does the segment (-1 -6 0 0 5 5).

Write a procedure Cross to elaborate two segments into one, two or three segments of the same union but with all intersections being empty.

Applying this procedure with the new segment and all preceding segments after each shift we obtain a presentation of Robot's trace as union of non-intersecting segments. The sum of their volumes yields the answer.

Task 4. Given a natural number $N \geq 2$. Let you to make N separate spexels. Mark N spexels in space by your will and apply the cutting operations (or 1-cutting ones). Find the minimal number of operations.

Example for 3*D*. Input: 3. Output: 8 cuttings (or 16 1-cuttings).

Solution. For cuttings, the answer is evident:

2D: $min\{U + V + 2: U*V \geq N\}$; *3D:* $min\{U + V + W + 3: U*V*W \geq N\}$.

For 1-cuttings the only way to solve is to examine all "compact" and "almost round" arrangements. It is difficult to prove that all arrangements that can be optimal are examined and to estimate complexity of solution. The jury is to prove it as a theorem (Pankov *et al.*, 2018) but the contestant can guess.

"Naturalness" means also the following. It is not necessary to write a program for generation of tests for a task. A human can compose sufficiently complex tests where the answer is "seen" but it is difficult to find it by means of any program.

Task 5. Given a natural number $N \geq 10$ and an arrangement of N spexels (in empty space), some of them are marked. How many, at least, other spexels are to be erased (or how many, at least, 1-cuttings are to be made) to pull out all marked spexels?

Example for 3D. Input: 175 voxels forming a $5\times5\times7$-parallelepiped, one central voxel is marked. Output: 2 voxels are to be erased (or 13 cuttings are to be made).

For 2D and $N < 200$ the answer in the test is seen by a human. For 3D and $N < 50$ observing of 3D-presentation of the test with semi-transparent cubes is seen. But a program would be sufficiently complex and to write it during bounded time is difficult (*).

Task 6. Given a natural number $N \geq 10$ and an arrangement of N spexels (adjacent spexels are supposed to be glued). Fix this arrangement. How many, at least, transparent spexels are to be added and glued to connect all given spexels?

Example for 2D (two pixels, the number N is less than 10).

Input: 2 \ 10 8 \ 11 2021. Output: 2013.

Solution would include the following items:

1) Unite given spexels into clusters.
2) Write a procedure finding the Manhattan distance between two clusters.

4. Tasks that Presenting Processes

The timexel is denoted as: *<spatial coordinate(s)>, <time>,* optionally *<color>.*

For example, the time step for the spectator is 1/60 second; one primary operation by the maker takes 10 microsecond = *tms*.

Task 7. Given $2N$ timexels: N timexels at time=1 and N timexels at time=2. All spatial coordinates are in 1..100, N in 2..10. A) all timexels are homogeneous or B) timexels at each time are colored equally; timexels cannot pass one through other.

How many operations (shifts by 1) are necessary to make this pass?

Examples for $N = 2$:

Example 7.1. A) 2D: Input: 2 \ 23 25 1 \ 28 22 1\ 25 23 2 \ 26 25 2. Output: 7.

Example 7.2. B) 3D:

Input: 2\ 8 23 25 1 blue \ 8 28 22 1 red \ 8 25 23 2 red \ 8 26 25 2 blue. Output: 9.

An example of task with (**).

Task 8. Given $N1$ colored timexels at time=1 and $N2$ colored timexels at time=2; timexels cannot pass one through other. A shift by 1 takes 1 *tms;* a repainting takes 2 *tms;* an erasing takes 3 *tms*; a creation of a colored timexel in any position takes 10 *tms.* ...

How many *tms* are necessary to make this transformation?

(If $N1 > N2$ then $(N1 - N2)$ erasings are necessary; If $N1 < N2$ then $(N2 - N1)$ creations are necessary; but sometimes one erasing and one creation are more profitable than many shifts; a similar note is right for repaintings).

A task on a sequence of events:

Task 9. Given natural numbers *M, N* and *K* and by *N* shining timexels for time = 1.. time = *K; N* < *M* < *K*N*. Timexels can turn on and turn off at each time. Find the minimum total number of shifts of *M* timexels to provide these events from their optimum initial arrangement.

Example for 1D:

Input: M = 3, N = 2, K = 3 \ 20 1 on \ 30 1 on \ 15 2 on \ 50 2 on \ 20 3 on \ 29 3 on Output: 11.

(the optimum initial arrangement is: M = 3 \ 20 1 on \ 30 1 on \ 50 1 off).

These tasks did not take into account the need to fix timexels in 3D.

Task 10. 2D (3D). Given *N* timexels at time = 1 and *N* timexels at time = 2. All spatial coordinates are in 1..100, *N* in 2..10. At time=1 all given timexels lean on the line *Y* = 0 (on the plane *Z* = 0) by means of pillars made of transparent timexels.

How many transparent timexels are necessary to be created to make this pass? (transparent timexels cannot move; some transparent timexels can be erased).

Example. 2D: Input: N = 3\ 4 2 1 \ 8 1 1 \ 4 7 1\ 5 2 2 \ 6 9 2 \ 9 4 2. Output: 8.

5. Conclusion

There are various publications on creation of new types of Olympiad tasks (Kemkes *et al.*, 2007), (Burton *et al.*, 2008), (Diks *et al.*, 2008), ours (Pankov *et al.*, 2018), (Pankov *et al.*, 2020). We hope that the proposed approach is new in general. We also hope that such tasks would enlarge the scope of tasks involved into Olympiads in Informatics and give ideas for young people to implement in hardware.

References

Dagienė, V., Skupienė, J. (2007). Contests in programming: quarter century of Lithuanian experience. *Olympiads in Informatics: Country Experiences and Developments*, 1**,** 37–49.

Pankov, P.S. (2008). Naturalness in Tasks for Olympiads in Informatics. *Olympiads in Informatics: Tasks and Training,* **2**, 16–23.

Pankov, P.S., Kenzhaliev, A.A. (2018) Combinatorial property of sets of boxes in Euclidean spaces and theorems in Olympiad tasks. *Olympiads in Informatics,* 12, 111–117.

Pankov, P.S., Kenzhaliev, A.A. (2020). Pattern Recognition and Related Topics of Olympiad tasks. *Olympiads in Informatics,* 14, 143–150.

Burton, B. A. Heron, M. (2008). Creating Informatics Olympiad Tasks: Exploring the Black Art. *Olympiads in Informatics: Tasks and Training*, 2, 16–36.

Diks, K., Kubica, M., Radoszewski, J., Stencel, K. (2008). A proposal for a task preparation process. *Olympiads in Informatics: Tasks and Training*, 2**,** 64–74.

Kemkes, G., Cormack, G., Munro, I., Vasiga, T. (2007). New task types at the Canadian computing competition. *Olympiads in Informatics: Country Experiences and Developments*, 1**,** 79–89.

P.S. Pankov (1950), doctor of physics-mathematics sciences, prof., corr. member of Kyrgyzstani National Academy of Sciences (KR NAS), was the chairman of jury of Bishkek City OIs, 1985–2013, of Republican OIs, 1987–2012, the leader of Kyrgyzstani teams at IOIs, 2002–2013, 2018–2020. Graduated from the Kyrgyz State University in 1969, is a head of laboratory of Institute of mathematics of KR NAS.

T.M. Imanaliev (1965), doctor of physics-mathematics sciences, professor of Applied Mathematics and Informatics Department at American University of Central Asia.

A.A. Kenzhaliev (1999). Bronze medal at IOI'2016. Student of Korea Advanced Institute of Science and Technology (KAIST).

Olympiads in Informatics, 2021 Vol. 15, 83–89
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.07

Extending Computational Thinking Activities

Zsuzsa PLUHÁR
Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary
e-mail: pluharzs@inf.elte.hu

Abstract. This report presents the newest extending activity idea, a challenge game of the Hungarian Bebras initiative. The goal of extension is to create unplugged computational thinking activities based on the Hungarian Bebras competition. The target group of this challenge game is flexible, and it could be used in the motivating phase of CT education, and as a first step in Computer Science education.

Keywords: Bebras, Hungarian Bebras initiative, computational thinking, unplugged activities.

1. Introduction – The Hungarian Bebras Initiative

1.1. *Bebras Initiative*

As Dagienė and Futschek (2008, 2012) summarized Bebras (“beaver”) initiative is an “informatics education community-building model” to motivate school students in the topic of Computer Science (CS) and computational thinking (CT). It works basically as an international informatics challenge and it is organized in more than 40 countries all over the world. It renewes our thinking about ICT, ICT education and thinking itself.

The initiative is based on a competition where we use “Bebras tasks”. They are short, motivating exercises that can be solved in 3 minutes, using thinking skills in the field of computer science, but they don’t require previous knowledge in that.

1.2. *Bebras in Hungary*

Hungary started to work in the Bebras initiative in 2010. Besides the main Bebras goals, the Hungarian organizers have country-specific goals, such as (1) showing how big, interesting and colorful the CS world is; (2) motivating children to be open to CS, problem

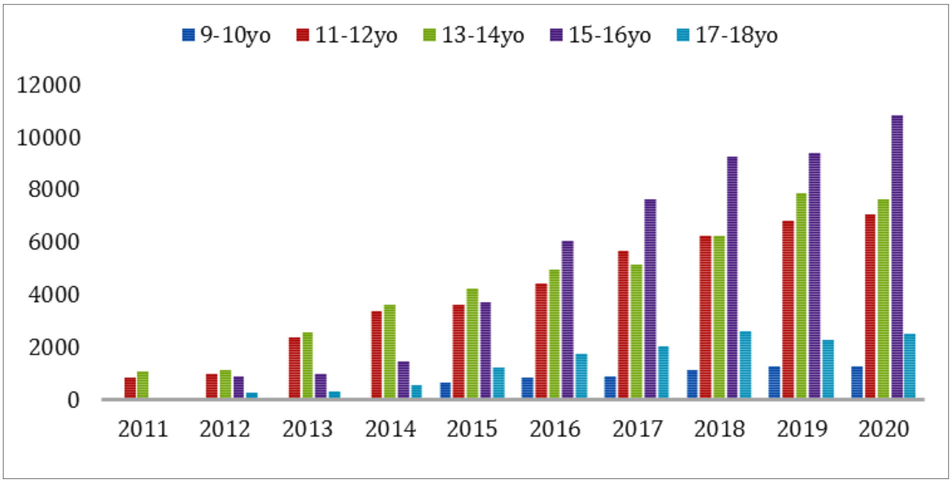


Fig. 1. The number of participants in the Hungarian Bebras challenge 2011–2020.

solving and thinking; (3) helping teachers make informatics education more colorful and understandable; (4) giving ideas to teachers for school and after-school activities.

The main basis of the initiative is the competition, the number of participants (Fig. 1) of which is growing continuously since it has started (Pluhár and Gellér, 2018; Pluhár, 2012).

Last year more than 29thousand students of ages 8–18 tested their knowledge in computational thinking.

2. Extending the Competition

In the Hungarian public education, only a small amount of Informatics is found, and the subject mostly focuses on how to use IT systems and software tools. Programming gets less emphasis. Project-based solutions can be used not only in classes but in after school activities, project weeks or days, school days, and summer camp activities.

To acheve the main aims of the Hungarian Bebras we need to extend the challenge and using unplugged, project-based activities.

In addition, the Faculty of Informatics at Eötvös Loránd University has courses where using activities to present and teach CT principles could be helpful. Bebras unplugged activities are applied not only in teacher education but also in the preliminary year of the international Computer Science BSc program (Pluhár and Torma, 2019).

The basic expandings use cards (Dagienė *et al.*, 2017) or small sheets with text and pictures. Creating project based-challenges should make use of several extended tasks, activities in a pack.

2.1. *Aims for Basis*

Our basic aims creating a pack are:

- The used tasks need to be funny, motivating and need to include as so many topics in CS as possible.
- The extension-activity needs to have a meaning and should be helpful to solve the problem.
- Each activity needs to be a physical activity: doing something with one's hands or body.
- The activities in a challenge-pack need to be varied and be motivating to solve.
- The tasks need to have the same attributes as in the basic challenge: participants don't need to have pervious knowledge to solve them, tasks can be solved in a few minutes, they need to be unambiguous, interesting, …
- The ideas can be changed: tasks can be used in modified order, it will be no problem if a task is removed or changed to another one.
- A teacher can prepare the used tools in the school, no need to buy big, expensive things. The cost should be low.
- The challenge is usable for several ages. The task contains guidelines to the solution that can be used by the coordinators.
- The game can be played by students and student-groups as families.

2.2. *Pilot*

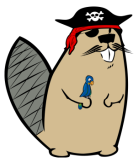
The first, pilot version was a pirate treasure game.

I prepared a treasure box with a padlock and 4 small Bebras tasks where participants had a number as the solution. If they added up the numbers they had the code to open the padlock.

I used the following tasks: Pirates (2015-IS-07), Quatris (2018-HU-04), Caesar coding (2017-CH-04b) and Lisas (2014-DE-04).

The main texts and problems were printed. I used printed and laminated cards and code wheels. The Pirates was used as a board game: I prepared a board and the pirate and the policeman were the game figures.

We tested the treasure game in university open days, Researchers' Nights, and on the event "Street of the Future". Our testers were between 6 and 70 years old. The treasures were small promotion gifts. The feedback was positive, and the fun-factor was always higher when a group tried to open the box. They could discuss the problem.



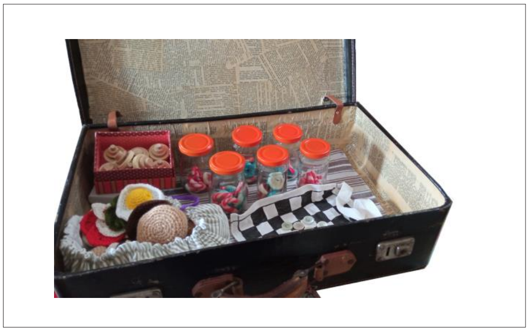


Fig. 2. The suitcase with activity tools.

3. The Unplugged Bebras Challenge Game

After the pilot treasure box project and small projects in the topic of escape games with Bebras tasks, I started to prepare a challenge game with guidelines.

A prepared suitcase contains the used tools with printed and laminated instructions and guidelines. The expanded tools are created as a DIY-project: the teachers (and students) themselves can prepare their own versions.

The basic challenge game is defined with 12 stations (activities), but in the pack, I prepared 15 tasks that can be used in different orders. We started developing more new activities that can be put into the suitcase (Fig. 2) and thus renew the game.

The students can participate in groups using a check-sheet where they can collect stamps and points. The game can be played in several versions: the collected points can depend on the rate of the help of the coordinator, the used time of solving the tasks, or the groups can get the stamp regardless of the time and help.

3.1. *Activities*

Each station has an activity that can be solved in a maximum of 10 minutes with support. Each station needs to have a coordinator who readies the station, gives small instructions and help, and resets the original state when a group leaves. The coordinators can tell stories about the task, its connections to CS and other disciplines or to real life.

The prepared guidelines of activities contain:

- The original state of the activity defined for age groups.
- Helping questions, instructions defined for age groups.
- Interesting facts in the topic of CS and connections with other disciplines or the real life.
- Ideas for variations with comments.

3.1.1. *Example Activity 1 – 2018-CZ-08c, Word Chain*

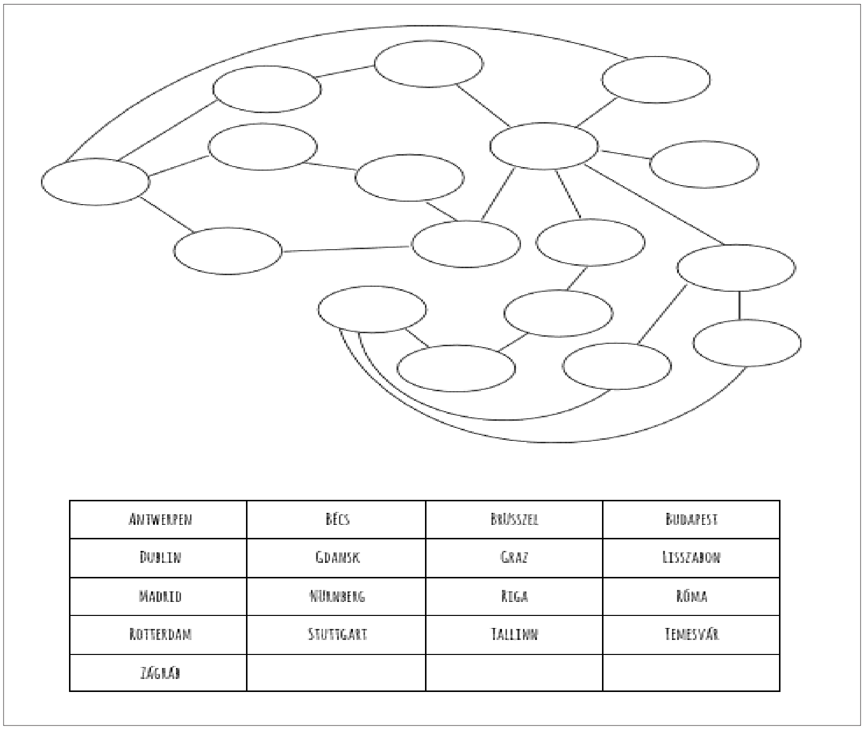
The computer science basis of this task is the generalized geography theorem. The original task asks for the longest word chan in a given graph.

This task has two parts in the activity. The first in our version is the same as the original task for the joungest two age groups. Other participants can play the game with words: they have a graph and words (names of cities) and they need to place the words into the graph based on the rules of the original game.

The second part is to write words in a given graph or to prepare their own game: create a graph with words and play.

In this activity the extension is a printed and laminated graph and the cut-out words. This is the easiest way of creating an activity – like a drag&drop application without an IT tool.

There are some activities where I supplemented the laminated papers with magnets, magnet boards or blutack, paper fasteners. In the case of board games the boards can be printed and the paws can be created manually.



ANTWERPEN	BÉCS	BRÜSSZEL	BUDAPEST
DUBLIN	GDANSK	GRAZ	LISSZABON
MADRID	NÜRNBERG	RIGA	RÓMA
ROTTERDAM	STUTTGART	TALLINN	TEMESVÁR
ZÁGRÁB			

Fig. 3. The template sheet to print, laminate and cut for the task 2018-CZ-08c.

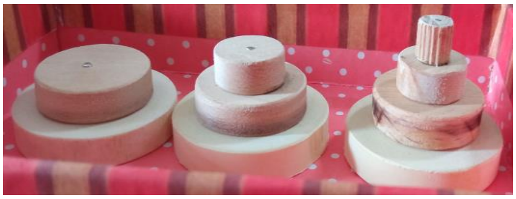


Fig. 4. The wooden disks used in task 2020-CH-04.

3.1.2. *Example Activity 2 – 2020-CH-04, Sudoku*

The original task is a combination of the sudoku and the skyscrapers. The size of the board and the given information (how many trees can be seen from a given position) define the levels for age groups. We prepared variations for the sudoku only – there are some trees on the board and participants/players need to place the others, or all the trees are on the board, and the players need to change their position to correct the situation based on the rules.

The basic board can be printed or cut from color paper. We used 4×4cm squares and wooden disks with different diameters placed on each other (Fig. 4).

For some activities we applied plasticine, crochet toys, yars and buttons.

The last planned activity is based on logic: the participants need to open metal mechanical puzzles, some of them were designed and created by my father.

4. Conclusion

The main aim of the Hungarian Bebras initative is the motivation. Both teachers and students are open to use new methods and ideas but they should be supported with more ideas and complex solutions.

After the testing phase we would like to prepare two versions from this challenge-suitcase: the first can be downloaded, and schools can create their own tools for activities. The second will be a suitcase that schools can buy and use during school activity days, project weeks or in after school events. This will be open for the Bebras community.

I would like to thank to all members of the International Bebras Community and give special thanks to the D-A-CH (Germany, Austria and Switzerland) members to not only help us prepare wonderful tasks but also to give ideas, help and support to organize the Hungarian Bebras Contests every year, and my father (✝) who supported and guided my ideas to solutions.

References

Dagienė, V., Futschek, G. (2008). Bebras International Contest on Informatics and Computer Literacy: Criteria for Good Tasks. In: R.T. Mittermeir, M.M. Syslo (Eds.), *Informatics Education – Supporting Computational Thinking*. Lect. Notes in Computer Science. Vol. 5090, Springer, 19–30.

Dagienė, V., Futschek, G. (2012). Knowledge construction in the Bebras problem solv-ing contest. *Conference: Constructionism*, 2012.

Pluhár, Z., Gellér, B. (2018). International Informatic Challenge in Hungary. In: *Teaching and Learning in a Digital World : Proceedings of the 20th International Conference on Interactive Collaborative Learning*. Berlin, Germany: Springer, pp. 425–435.

Pluhár, Zs. (2012). Bit HÓDítás. In: Ollé János (ed.) 4. *Oktatás-Informatikai Konferencia: Tanulmánykötet*. Budapest. ELTE Eötvös Kiadó, 187–191.

Pluhár, Zs., Torma, H. (2019). Introduction to Computational Thinking for university students. In: *Lecture Notes in Computer Science*, 11913. pp. 200–209, 10 p.

Dagienė, V., Futschek, G., Koivisto, J., Stupurienė, G. (2017). The card game of Bebras-like tasks for introducing informatics concepts. In: *ISSEP 2017 Online Proceedings*. Helsinki, 13.11.2017–15.11.2017.

Zs. Pluhár is an assistant lecturer of the Faculty of Informatics at Eötvös Loránd University, Budapest, Hungary. She is a member of the T@T (Technology Enhanced Learning) Lab and works mostly in teacher education. She is the head of the Professional Community of Public Education at John von Neumann Computer Society. Her research fields are computational thinking, education of robotics and STE(A)M. Since 2011 she has been organizing the Bebras informatics contest in Hungary.

Olympiads in Informatics, 2021, Vol. 15, 91–104
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.08

Teaching and Examination Process of Some University Courses before vs during the Corona Crisis

Vesna Dimitrievska RISTOVSKA, Emil STANKOV,
Petar SEKULOSKI

Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University
st. Rugjer Boshkovikj 16 Skopje, Macedonia
e-mail: vesna.dimitrievska.ristovska@finki.ukim.mk, emil.stankov@finki.ukim.mk, petar.sekuloski@finki.ukim.mk

Abstract. The newly emerged corona crisis in our country, but also much broader – on the entire planet, caused by the pandemic scale of COVID-19 virus, dictated the need for adjustment of the teaching and examination process of many university courses. At our institution, Faculty of Computer Science and Engineering (FCSE) in Skopje, starting from March 17, 2020, until today (March 2021), classes and exams are completely realized through distance learning systems, i.e. using the BigBlueButton video conferencing system, implemented in the Courses and Exams student services – the official FCSE websites on the Moodle e-learning platform.

For all faculty courses, lectures, auditory and laboratory exercises, colloquia and exams, all take place via a video conferencing system for distance education.

In this paper we present a comparative analysis of the conduction of some courses at FCSE in classical conditions, as opposed to the conditions with distance education. We have considered the analysis mainly from the aspect of the approach to teaching, as well as from the aspect of exam conduction and achieved exam results. The analysis of those aspects leads us to conclusions about several positive and negative sides that we noticed in distance education compared to the classical conditions of classes and exams. Our findings also may apply on the organization of online contests, especially in informatics.

Keywords. distance education, online classes, online exams, university subjects, corona crisis, BigBlueButton (BBB), Calculus, survey.

1. Introduction

In this paper we review and compare some aspects of the teaching process as well as the exam results for several courses at our institution, Faculty of Computer Science and Engineering (FCSE), at the Ss. Cyril and Methodius University in Skopje, in the period before the corona crisis, as opposed to the first year of the corona crisis. Electronic services, such as announcements regarding classes and exams, exam registration, exam surveys, submission of solutions to assignments, projects, etc., as well as exams conduction, have been used at FCSE for many years. Experiences with the usage of electronic services have made it much easier to bridge the interruption of the classical teaching process caused by the corona crisis. The teaching program of most subjects at FCSE includes theoretical lectures, auditory and laboratory exercises. In regular circumstances, the lectures and auditory exercises typically take place in the classroom, using a whiteboard and a video beam. The emerging corona crisis in our country, and much broader - on the entire planet, caused by the pandemic of the virus causing the COVID-19 disease, dictated the need for adjustment of the teaching and examination process of many university courses.

The World Health Organization (WHO) declared a pandemic on March 11, 2020, and less than a week after that date, i.e. as of March 17, 2020, until today (March 2021), the classes and exams at FCSE are being conducted through distance learning systems completely. This is done using the BigBlueButton (BBB) video conferencing system, implemented in the Courses and Exams student services, i.e. the official FCSE websites on the Moodle e-learning platform.

Distance learning is not a modern phenomenon. In the twentieth century, distance learning includes technologies such as radio and television, but without mass usage. As the technology was advancing in the new computer era, this way of learning emerged to new multimedia technologies (Pregowska *et al.*, 2021). Although, many of the educational programs partially included some distance learning methods, the main challenge came with the COVID-19 pandemic (Chan *et al.*, 2021).

Section 2 discusses the related work on distance education throughout the ongoing pandemic period, on different educational levels. In the following sections we present our comparison of the teaching process and the results of the exams for several courses at FCSE.

2. Related Work

Asgari *et al.* (2021) conducted an observational study at California State University, Long Beach (one of the largest and most diverse four-year university in the U.S.) to enhance engineering online education during the pandemic. From six engineering departments, a total of 110 faculty members and 627 students answered questions in their surveys, to highlight the challenges they experienced during the online instruction in Spring 2020. Their results show different issues that negatively influenced the online engineering education: for example, more than half of the students indicated lack of engagement in class, difficulty in maintaining their focus and Zoom fatigue after attending multiple online sessions. The authors recommend strategies for educational stakeholders to fill the tools and technology gap and improve online engineering education.

Dhawan (2020) focuses on the importance of online learning and the Strengths, Weaknesses, Opportunities & Challenges (SWOC) analysis of e-learning modes in the time of crisis. This article observes the growth of EdTech Start-ups during the time of pandemic and natural disasters and suggests how to deal with challenges associated with online learning.

Ballova Mikuskova and Veresova (2020) focus on distance education implemented in primary and secondary schools in Slovakia during the COVID-19 pandemic. They analyze distance education from teachers' point of view, particularly from the aspect of the connection between the teacher's previous experience, perception and the management of the educational process in the pandemic period.

3. Comparative Analysis of Regular Classes Vs Classes during the Corona Crisis

Most of the subjects at FCSE involve classes, which consist of lectures, auditory and laboratory exercises. The classical way of conducting the lectures and exercises (until the onset of the corona pandemic) was in the classroom, using a whiteboard and a video beam. However, as explained previously, due to the COVID-19 pandemic, the process had to be adjusted to the newly emerged circumstances.

During the corona crisis, for all courses at the faculty, distance education realized through lectures, auditory and laboratory exercises, as well as colloquia and exams, took place through the BigBlueButton (BBB) video conferencing system, implemented in the Courses and Exams student services, i.e. the official FCSE websites on the Moodle e-learning platform.

In each of the following subsections we briefly describe the classical teaching approaches for the theoretical lectures, auditory and laboratory exercises, and then elaborate on the adjusted online approaches that were implemented with the outbreak of the pandemic.

3.1. *Theoretical Lectures*

In the classical theoretical lectures for the courses at FCSE, the teacher typically explains the planned topic, uses video-beam presentations for the lesson, which show definitions, algorithms, program sequences, schemes, theorems, proofs, properties, as well as tasks from the material. Depending on the material under consideration, the teacher writes solutions to problems in detail on the whiteboard or on his computer – while sharing the computer screen using a video beam, sketches graphs or diagrams, derives proofs of theorems, etc.

At the very beginning of the corona period, several questions arose about the manner of teaching, for which a decision had to be made as soon as possible, in order to continue with the interrupted educational process:

1. Should the lectures be recorded, or should each lecture be conducted online only, without recording, at a time provided in the faculty schedule of classes?

2. If the lectures are recorded, should it be done beforehand, and not at the class itself, in order to prepare a better and cleaner recording, on which there would be no possible questions from students?

According to the strategy adopted by the Teaching/Scientific Council of the faculty, the details of the conduction of online classes for each course were left to the estimate of the course teachers. Thus, there were differences in the teaching approaches adopted in different courses.

3.2. *Auditory Exercises*

The classical teaching approach for the auditory exercises of most subjects at FCSE is such that the teaching assistant explains and writes the solutions of the topic's tasks on a whiteboard or on his computer – while sharing the computer screen using a video beam, sketches graphs, tables, ideas, etc. Very often, a parallel display of the material on a video beam is used during the classes, so that the students can have a better insight into the textual description of the tasks.

3.3. *Laboratory Exercises*

On the other hand, the regular laboratory exercises involved solving basic and advanced tasks related to the subject's material in some appropriate software environment / programming language, such as C++, Python, R, GeoGebra, Wolfram Mathematica, etc. Until two years ago, the presence of students at the laboratory exercises was mandatory, but for the last two years, until the onset of the corona pandemic, these exercises were mostly consultative. Namely, every week a manual for the exercise was published for the students, with appropriately explained commands and solved tasks, but also tasks assigned for individual work of the student intended to prepare him / her for the lab exam. The student had the possibility to attend the laboratory and ask the responsible teaching assistant about the planned tasks, but he / she could also work out the planned material for the laboratory exercise at home.

With the outbreak of the pandemic and the caused interruption of classes, similar issues regarding the way of continuation of the educational process arose to those described for the theoretical lectures:

1. Should the auditory and lab exercises be conducted through online lessons using a webcam or by sharing the computer screen and solving tasks directly in a software environment, or on a piece of paper with a pen, or a digital board, or using pdf materials and additional oral explanation?
2. Should the auditory and lab exercises be conducted using pre-prepared videos, which could be used during the scheduled terms for the auditory and lab exercises, so that the online classes would be used only for clarification of some of the details, or only some of the tasks according to the students' needs?

Once again, according to the official strategy of the faculty, the decision on the approach to classes realization was left to the teaching stuff of each individual course, and thus, different approaches were taken for different courses.

In the next section we analyze the exams conduction for some courses at FCSE: the classical approach (before the corona crisis), as opposed to the online approach implemented during the corona crisis.

4. Comparative Analysis of Regular Exams Vs Exams during the Corona Crisis

4.1. *Exams before the Corona Crisis*

In regular circumstances, before the outbreak of the pandemic, the exams for most courses at FCSE were conducted in a classroom or amphitheater, so that students wrote answers to questions and solved tasks on paper. Exception was the lab exam, which was taken in the laboratory, on a computer, by solving tasks in the appropriate programming language.

4.2. *Exams during the Corona Crisis*

The first summer midterms, which before the corona crisis would normally take place in April, in the summer semester of 2019/2020 were held only for some courses, the so-called "pilot courses" – mostly with a smaller number of students.

The following three major exam sessions: June, September and January, as well as the two partial exams (colloquia) in the winter semester of 2020/2021 were held completely online, usually using two cameras, through the BigBlueButton (BBB) system. One camera was from the student's computer (the student was required to share the screen for constant insight into his / her work), and the second was a mobile phone or web camera, positioned to the right of the student, so that the student, his / her desk and the computer he / she is working on could be followed closely. Any materials other than those permitted for the course were prohibited during the examination. The use of social networks or communication software was also strictly forbidden.

For students who would not have the appropriate technical conditions for taking online exams from home (adequate hardware equipment and stable Internet connection), the faculty provided the option of taking exams with physical presence. Such students were required to register by filling a survey in a timely manner prior to the exam session, and all exams in that session were to be taken in person. For the needs of these exams with physical presence, in each exam session, an appropriate, large enough computer laboratory was prepared in which a safe distance was provided between the students and all recommendations and measures for protection of students' health were respected. Furthermore, a high-quality webcam was installed in the laboratory in order to provide a constant supervision during the exam. For these students, as well as for the other students present at the exam, the same rules for taking the exam applied.

For each online exam, students are required to connect to an assigned BBB session. Typically, about 20 students are assigned per BBB session. All activities are being recorded throughout the entire BBB session. Before the start of the exam, each student is identified with his / her student ID or other personal identification document. If the student uses one or more sheets of paper as an aid in solving the exam questions / tasks, he / she must show it on camera before the beginning of the exam so that the teacher can be assured that it is empty. Each teacher follows all students with video and audio signal during the exam, and on the teacher's screen there is a list of names of all students currently present at the BBB session. If necessary, the teacher can zoom in on a student's recording to check if there is a suspicious activity. If a student has a question, he / she can ask it in written (electronic) form in the public chat of the BBB session.

All course teachers, course teaching assistants, and additional teaching stuff responsible for the exam supervision, are connected to a viber group for faster mutual coordination on exam questions and other exam issues.

4.3. *Problems Part of the Exams during the Corona Crisis*

In the June exam session, the Problems part of the exam for most of the courses at FCSE was realized directly in an appropriate software environment, except for the Calculus 1 and Calculus 2 courses, where students were required to solve the tasks on paper using a pen. The duration of this part was 2.5 hours, and at the end of the exam the students were obliged to take photo(s) of the solutions using a mobile phone and attach them at the appropriate assignment on the Moodle system.

From the September session onwards, the Problems part in Calculus 1 and Calculus 2 was conducted as follows: the student solves the tasks on paper with a pen, and then writes the solutions in a math formula editor, based on TEX-commands, which is built-in in the Moodle e-learning platform. The duration of the exam was increased to 3 hours in order to enable students enough time for typing the solutions in the editor. Prior to the September session, the teaching assistants had prepared a tutorial for the students, with ready-made examples to illustrate how to write the basic required formulas in the editor.

4.4. *Theoretical Part of the Exams during the Corona Crisis*

The Theory part of the exam, took place online, through a Moodle quiz which contained different types of questions: some of the questions were single or multiple choice, for some of them – a short answer had to be entered (for example – a number or a letter), and for some – the student had to write a definition, property, proof, etc.

Depending on the teacher's findings, if necessary, some of the students were called for additional oral interrogation in front of the camera in order to assess their knowledge more precisely.

4.5. *Lab Exams during the Corona Crisis*

This part of the exam took place on a computer, with two cameras on, while the student solved several tasks in some appropriate programming language.

5. Case Study: Comparison of Exam Results from Some Courses before VS during the Corona Crisis

5.1. *Calculus 1 – Fall/Winter (October to February) Semester 2019/2020 and 2020/2021*

Table (see Fig.1) presents an analysis of the results from the partial exams and the January exam session, for the Calculus 1 course, in the winter semesters of the academic years 2019/2020 and 2020/2021. From the data given in Fig. 1, we can conclude that there is an increasing number of students that pass the Calculus 1 course through partial exams during

Academic year	**2019/2020**	**2020/2021**
Number of students that pass Theory part on partial exams	51 / 202	68 /104
Percentage of students pass Theory part on partial exams	25.25	48.57
Number of students that pass Problems part on partial exams	42 / 188	44 /90
Percentage of students pass Problems part on partial exams	22.34	48.89
Number of students that pass the exam on partial exams	**36**	**40**
Percentage of students that pass the exam on partial exams	**19.15**	**44.44**
6 (E)	11	15
7 (D)	3	8
8 (C)	10	7
9 (B)	10	5
10 (A)	2	5
Number of students that pass Theory part in January session	51 / 121	22 / 33
Percentage of students pass Theory part in January session	42.15	66.67
Number of students that pass Problems part in January session	39 / 114	25 /46
Percentage of students pass Problems part in January session	34.21	54.35
Number of students that pass the exam in January session	**31**	**11**
Percentage of students that pass the exam in January session	**27.19**	**33.33**
6 (E)	16	7
7 (D)	8	3
8 (C)	5	0
9 (B)	2	0
10 (A)	0	1
Total # students that pass the exam (part. Exams + January session)	**67/239**	**51/112**
Total percentage of students that pass the exam (part. Exams + January session)	**28.03**	**45.54**

Fig. 1. Analysis of partial exams and January exam results in 2019/2020 and 2020/2021 (online).

online examination in 2020/2021, from 19.15% to 44.44%. For the number of students that pass the exam in January, we can say that there is no significant increase. Also, from Fig. 2, we can say that there is an increase in the number of students that achieved minimum points for passing the exam for the different exam parts during online examination.

The distribution of grades in percent is shown in Fig. 3. We can say that there is an increase in the percentage of the grades 10 (A), 6 (E), and 7 (D), and decrease of the grades 8 (C) and 9 (B).

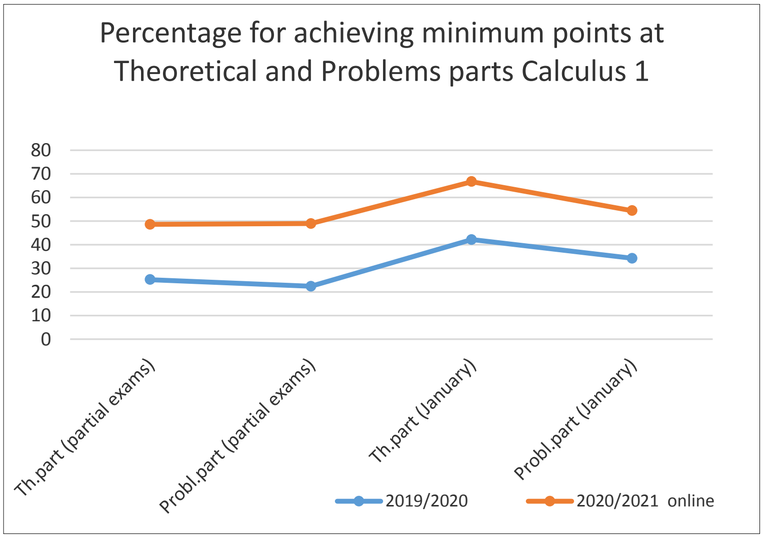


Fig. 2. Percentage of students that achieved minimum points on the different exam parts for Calculus 1.

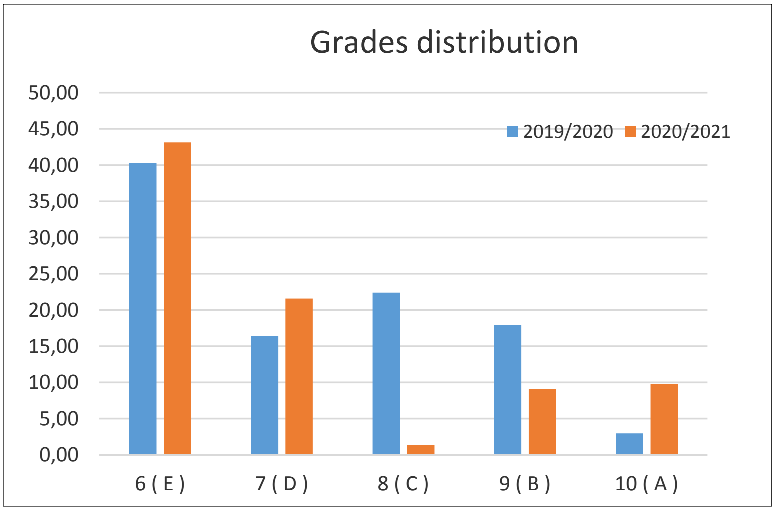


Fig. 3. Grades distribution for Calculus 1.

5.2. *Comparing Exam Results: Business Statistics, Calculus 1, Calculus and Calculus 2*

In Fig. 4 and Fig. 5, it is obvious that the percentage of students that pass the course Calculus 2 through partial exams in 2019/2020 is larger than the other given percentages. The reason for this can be the new program that was accredited that year. All the students enrolled in that course were students that are enrolled in Calculus 2 for the first time.

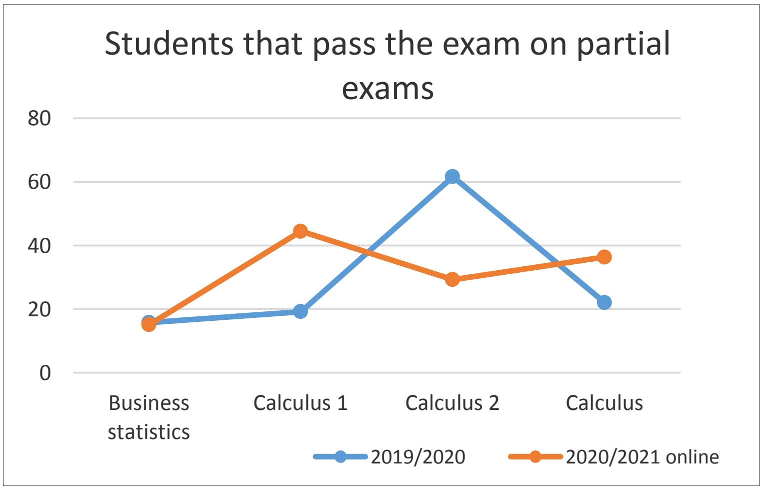


Fig. 4. Percentage of students that pass the partial exams for different courses.

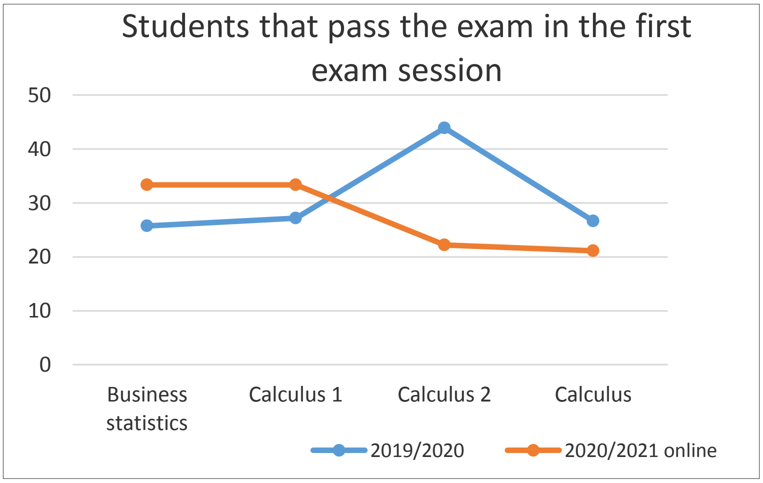


Fig. 5. Percentage of students that pass the first exam for different courses.

6. A Survey for the Online Classes during The Corona Crisis

For the purpose of this paper we conducted a survey on the online practices in teaching and learning for the course Calculus 1. The survey was taken by students that are enrolled in Calculus 1 in the academic year 2020/2021 (during the corona crisis). The questionnaire was composed of 8 questions that would detect advantages and disadvantages of the learning process. The questions were:

1. How satisfied are you with the online classes (theoretical lectures and auditory exercises):
   a. Very satisfied.
   b. Poorly satisfied.
   c. Neutral.
   d. Unsatisfied.
   e. Very unsatisfied.
2. If you have noticed, please list at least one positive side of online classes that you think is important for mastering the material and passing the exam.
3. Please, if you have noticed, list some disadvantages of the online classes.
4. Please, if you have noticed, list some disadvantages of the online exams.
5. During the online lectures, the quality of the audio signal was:
   a. Very good.
   b. Good enough.
   c. Good, with little noise or interference.
   d. Bad.
   e. Very bad, barely understandable.
6. During the online auditory exercises, the quality of the audio signal was:
   a. Very good.
   b. Good enough.
   c. Good, with little noise or interference.
   d. Bad.
   e. Very bad, barely understandable.
7. During the online auditory exercises, the quality of the video signal was:
   a. Very good.
   b. Good enough.
   c. Good, with little noise or interference.
   d. Bad.
   e. Very bad, barely understandable.
8. If, when using a recorded material from this course, you come across an obscure part, then you seek help from:
   a. A colleague.
   b. A friend.
   c. A professor.

d. A teaching assistant.
e. By sending message via e-mail to a professor/teaching assistant.
f. In the scheduled online office hours of the course's teaching stuff.
g. A book (hard copy).
h. Other online resources on the Internet.

From the results of the survey, the lectures and exercises satisfied the students in the technical sense, such as the quality of video and audio signals, the usage of open platforms for e-Learning, etc. The main advantage mentioned in this survey is that video recordings of the lectures and exercises are available to the students at any moment in time. Also, some of the students think that they have better organization for their study time.

The main disadvantage detected is that during the online classes there is no "eye-to-eye" contact with the lecturer. Also, 95% of the students find the lack of direct communication with other students as a main pitfall of online classes.

As a main disadvantage of the examination process, the students highlight the process of answering questions in the special formula editor. This process takes more time than answering with pen and paper. Also, some of them mention technical issues with their Internet connection during the exams.

Some interesting results were obtained for the question 8 of the survey. These are shown graphically in Fig. 6. As we can see, when students have problems with the material, most of them seek help either from a colleague, from a book, or from other online resources on the Internet.

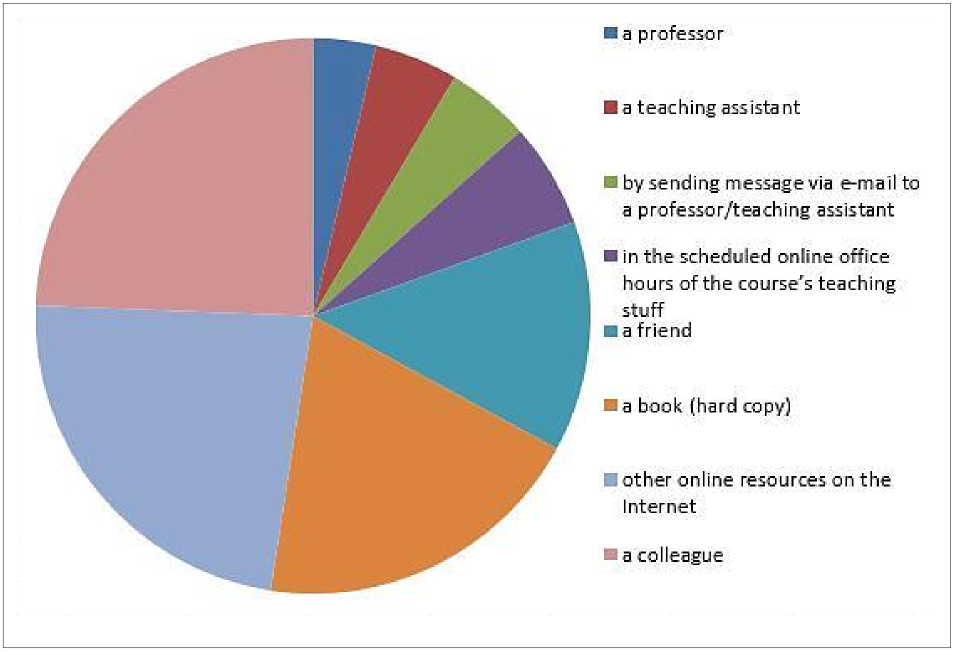


Fig. 6. Results for the question 8 of the survey on the Calculus 1 online classes.

7. Conclusion

Overall, there is some increase in the percentage of students who have passed during the COVID-19 period, depending on the subject. The exception here is Calculus 2, where in 2018/2019 there was a newly accreditated program and there were no students retaking the exam, in contrast to 2019/2020 where there were a significant number of students retaking the subject, which has a significant impact in the reduction of the passing rate of the colloquia and the first exam session.

The most lacking aspect of teaching during the corona crisis is the "eye-to-eye", or direct communication between the students themselves, as well as between the students and the teaching staff.

From the results of the survey, the lectures and exercises satisfied the students in the technical sense, such as the quality of video and audio signals, etc. The main advantage mentioned in this survey is that video recordings of the lectures and exercises are available to the students at any moment in time. Also, some of the students think that they have better organization for their study time.

The main disadvantage detected in the survey is that during the online classes there is no "eye-to-eye" contact with the lecturer. Also, 95% of the students find the lack of direct communication with other students as a main pitfall of online classes.

The software and technical support that we have employed in the first year from the beginning of the corona crisis at FCSE, gave satisfactory results and experiences, which could be applied in the implementation of competitions in informatics, as well as other competitive disciplines. The emphasis from the experiences is on the online identification of the student, as well as the monitoring of his work for the entire task solving time.

Here we want to emphasize that one of the most important links in the success of the online exam conduction is the provision of technical support for a stable Internet connection during the examination process.

As the COVID-19 pandemic continues for a second year in a row, sharing our results in this study can help with more effective planning and choice of best practices to enhance the efficacy of online and distance education during COVID-19, as well as post-pandemic.

Acknowledgement

The research presented in this paper is partly supported by the Faculty of Computer Science and Engineering, at the Ss. Cyril and Methodius University in Skopje.

References

Asgari, S., Trajkovic, J., Rahmani, M., Zhang, W., Lo, R. C., and Sciortino, A. (2021). An observational study of engineering online education during the COVID-19 pandemic. *PLoS ONE* 16(4): e0250041.

Ballova Mikuskova, E., and Veresova, M. (2020). Distance education during Covid-19: the perspective of Slovak teachers. *Problems of Education in the 21st century*, 78(6), 884-906.

BBB – BigBlueButton, Open source virtual classroom software.
`https://bigbluebutton.org/` (accessed 16/5/2021)

Chan, R. Y., Bista, K., and Allen, R. M. (2021). *Online Teaching and Learning in Higher Education during COVID-19: International Perspectives and Experiences*, First Edition.

Courses – The course management system of the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje.
`https://courses.finki.ukim.mk/` (accessed 16/5/2021)

Dhawan, S. (2020). Online Learning: A Panacea in the Time of COVID-19 Crisis. *Journal of Educational Technology Systems*, 49(1), 5-22.

Exams – The exams management system of the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje.
`https://ispiti.finki.ukim.mk/` (accessed 16/5/2021)

GeoGebra – The mathematical software.
`https://www.geogebra.org` (accessed 16/5/2021)

Pregowska, A., Masztalerz, K., Garlinska, M., and Osial, M. (2021). A Worldwide Journey through Distance Education—From the Post Office to Virtual, Augmented and Mixed Realities, and Education during the COVID-19 Pandemic. *Educ. Sci. 2021*, 11(3), 118.

WHO – World Health Organization.
`https://www.who.int/` (accessed 16/5/2021)

Wolfram Mathematica: Modern Technical Computing.
`https://www.wolfram.com/mathematica/` (accessed 16/5/2021)

V.D. Ristovska is an associate professor at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, in Skopje. Her research interests include uniform distribution of sequences, numerical analysis, optimization methods, and topological data analysis.

E. Stankov is a teaching and research assistant at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, in Skopje. He is a member of the Executive Board of the Computer Society of Macedonia and has actively participated in the organization and realization of the Macedonian national competitions and Olympiads in informatics since 2009. Currently he is a Ph.D. student at the Faculty of Computer Science and Engineering. His research includes analysis of program code correctness using different techniques, and its application to e-learning.

P. Sekuloski is a demonstrator (younger teaching and research assistant) at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, in Skopje. Currently he is a M.Sc. student at the Faculty of Computer Science and Engineering. His research includes Topological Data Analysis, Machine learning and application of some method from algebraic topology to computer science.

Olympiads in Informatics, 2021, Vol. 15, 105–118
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.09

Algorithmic Thinking and New Digital Literacy

Marina S. TSVETKOVA, Vladimir M. KIRYUKHIN
Russian Academy of Natural History, Russian Federation, Russia, Moscow, 105037, box 47
e-mail: vkiryukh@gmail.com , ms-tsv@mail.ru

Abstract. The article discusses the concept of algorithmic thinking in the context of the history of the formation of school informatics, in the competencies of new digital literacy and in the system of developmental education. The structure of digital literacy based on algorithmic supports is shown and an example of an educational template of algorithmic tasks for younger students is given.

Keywords: digital literacy, school informatics, system of developmental education, algorithmic thinking, primary general education.

1. Introduction

The development of algorithmic thinking in children in the 21st century is an integral part of modern digital literacy, as the ability to learn and develop in a digital environment. In the digital age, it is algorithmic thinking that becomes the intellectual basis for working with digital information to include a person in the modern world of information systems and digital communications. Artificial intelligence, control of machines and digital services in life, study, work, and creativity are of particular importance here.

The world of the digital age is artificially created by people, it is filled with digital devices and programs that make up the services and resources of the digital environment. The challenges of this world are new digital literacy, the competencies of which are based on fundamental knowledge and skills in the subject of informatics. This is algorithmic knowledge and ability to use programs and digital devices, as well as creativity for the implementation of algorithmic ideas in information systems and robotics.

New digital literacy is becoming an integral part of traditional literacy from early childhood for children living in digital environments from birth. And school informatics is a subject that forms digital literacy based on algorithmic thinking. Algorithmic thinking of a person living in a dynamic digital environment affects the duality of the content of informatics – these are the scientific and technological foundations of the subject. Science gives rise to all new digital instruments, devices, systems. New digital technologies stimulate the search and development of new algorithms, models and information structures.

2. Programming is the Second Literacy

During the formation of the subject of Informatics in the 1970s the Russian scientist Academician A.P. Ershov developed a school informatics course program (Ershov, 1988). He also put forward the thesis "**programming is the second literacy**" (Ershov, 1981) and this thesis is still relevant now.

Professor S. Papert also predicted an all-round intrusion of the computer into the world of the child, when the computer would become an intellectual tool used by the child with the same immediacy with which he uses a pen and pencil, but with much greater variety. Interpreting Piaget's observations that the child makes most of his intellectual discoveries on his own, provided that the background surrounding him is sufficiently rich, Professor S. Papert showed that the computerization of this background will create a new, unprecedented operating environment that will require new ideas in developmental psychology.

A. Ershov noted: "If the development and spread of typography has led to universal literacy, then the development and spread of computers will lead to universal programming skills. … It's not about imposing new skills and knowledge on children, but about manifesting and formulating those aspects of thinking and behavior that really already exist, but are formed spontaneously, unconsciously".

It is important that new knowledge and skills are formed in the world of programs: "The world of programs is far from just the filling of computer memory. First of all, this is a huge stock of operational knowledge accumulated by mankind and now only actualized by computers, robots, and automatic devices. An even larger stock of programs is stored in the gene pool of all living things: its decryption and use are largely the subject of biology and its new sections, including genetic engineering" (Ershov, 1981).

A. Ershov also noted the importance of developing these new skills from an early age. "The question of how to teach children the ability to plan their actions and their consequences, what kind of operating environment is needed in this case, is very far from those methodological alternatives that we are discussing, for example, in professional training in programming. On the one hand, we must make this environment natural for the child, on the other hand, it must be rich enough so that he could, as psychologists say, create a theory of a cognizable phenomenon on his own ... The laws of programming, the laws of information processing appear in the form operational rules reflecting the direct experience of humankind ... This is an important statement about the deep and indissoluble connection of operational knowledge and algorithmic thinking with other components of education" (Ershov, 1981).

As a result, the inclusion of algorithmic thinking and programming technology in the fundamentals of the subject of Informatics was recorded. And the subject of computer science itself over the years of development has become a guarantor of the formation of new digital literacy in schoolchildren on the basis of algorithmic thinking, which is reflected in the Russian school educational standard.

3. The Role of Algorithmic Thinking in the Developmental Education System in the Digital Age

In the 20th century, a system of developing education was developed and implemented in Russian schools. This made it possible to place at the center of learning not the process, but the child, his success in development on the basis of educational activities. The basis of activity is the concept of actions, which consist of operations, that is, ways of performing actions. This concept is most fully reflected in the psychological theory of activity, which is associated with the names of Soviet scientists L.S. Vygotsky (Vygotsky, 1934), A.R. Luria (Vygotsky and Luria, 1930), P.P. Blonsky (Blonsky, 1935), S.L. Rubinshtein (Rubinshtein, 1989), P.Y. Galperin (Galperin, 1985) and is described in the book by A.N. Leontyev "Activity. Consciousness. Personality" (Leontiev, 1975).

Algorithmic thinking is based on the system-activity approach, in fact it is operational and includes various approaches based on research by scientists: structures and operations of thinking (Piaget, 1985), computational thinking in cooperation with a computer (Papert, 1988), algorithms and programs in the structure of thinking (Ershov, 1988), action planning and advanced development (Zankov, 1975), research algorithmic approach in operational thinking, thinking operations (Shapiro, 1975), modeling, logical thinking (Elkonin, 2007; Davydov, 1995; Davydov, 1996), interiorization as a process of transforming external, objective actions into internal mental ones, exteriorization – the transition from an internal, mental plan of action to an external one, implemented in the form of techniques and actions with objects (Galperin, 1985).

The essence of the algorithmic approach is that the student is taught not only the concepts of the essential properties of certain objects, but also the rules by which these properties are associated with the actions necessary to solve certain problems (algorithms).

The most important principle of algorithmic thinking is its effectiveness at the level of a mental image of a decision. The key to the disclosure of creative activity in productive activity based on algorithmic thinking is **anticipation** – the presentation of the result of an action in the mind of a person before it is actually carried out. The computer as a tool for the implementation of algorithmic ideas with feedback has created completely new conditions for the development of algorithmic thinking in children, when the computer becomes a child's assistant for productive activity and creativity, removes the barrier in the formation of an internal image of a solution and the way to achieve it based on the experience of obtaining feedback – a computer sketch, a digital prototype created by him or selected from digital templates.

4. Intellectual and Instrumental Pillars of Algorithmic Thinking

Algorithmic thinking has intellectual and instrumental support, that is, it has dualism: algorithmic knowledge as a formalized method of operational thinking, on the one hand, and algorithmic skills as a tool for implementing algorithmic ideas in practice, on the other hand. These principles of algorithmic thinking are central to the educational process based on algorithmic problems for children.

The intellectual (theoretical) pillars of algorithmic thinking are reflected in the mathematical foundations of informatics, represented by intelligent algorithmic constructs, such as: numbers and coding, digital information, the order of calculations, patterns, combinatorics, probability, sets, logic, information presentation, lists, graphs, relationships, commands and algorithms for the executor, program control real and virtual command executor.

Instrumental (practical) supports ensure the use of ready-made programs, tools for implementing algorithms in a digital environment and are reflected in the technological foundations of informatics in the form of instrumental algorithmic constructs, such as: the computer as a command executor; computer devices, processor and internal memory; program principle of computer control; device management; the computer as a command executor; control commands; program; menu in the program; computer tools for information processing; setting up tools; programming; command executor; automation of program control; net; Internet – an environment for control of information; artificial Intelligence; smart technology.

From these algorithmic constructs base digital literacy competencies are formed.

5. The Structure of Digital Literacy: The Digital Triangle

In fact, algorithmic thinking led humanity to the invention of the computer, which is a universal device for implementing algorithms in the form of programs. We can say that algorithmic thinking is the foundation of digital literacy. Intellectual (internal) and instrumental (external) supports of algorithmic thinking form new digital competencies of children only in their unity.

The structure of digital literacy includes three groups of key digital competencies accumulated over decades of development of the digital world: computer literacy, information literacy and communication literacy. Computer literacy has formed a group of digital literacy technical competencies based on a variety of digital devices and programs, automation tools. Information literacy has formed the competence of working and understanding digital information based on a variety of ways of presenting it, including media and virtual reality. Communication literacy was formed on the basis of the globalization of digital communications, including cloud, mobile technologies.

These three groups of digital competencies make up the digital literacy competency triangle. Each group has a typical set of educational actions: intellectual (know, un-

Table 1
The digital literacy framework

Key competencies	Intellectual competencies (know)	Instrumental competencies (be able to)	Social competences (act)
Information competencies (working with digital information)	Types of information and algorithmic bases for working with information. Methods for presenting digital information. The specifics of information processes in the digital environment. The features of computer mo-dels of objects and processes of various nature	Work with information and information structures. Process information of various types on a computer. Use smart tools to apply and create algorithms, models, and programming.	Work with information to control various devices. Self-adapt to new ways of working with information, such as: multimedia, virtual reality, simulators, robotic devices and robotic programs, cyberworlds in everyday life and public space
Technical digital competencies	The composition of the com-puter and the purpose of digital devices. The programmatic principle of the computer. Variety of digital devices and programs.	Use various interfaces to control digital devices. Work on a computer in different software environments.	Apply tools for working with new digital devices. Get involved in working with digital information in different and new software environments independently.
Digital communications	Composition and principles of operation of computer networks and the Internet. The structure of the presentation and transmission of information in the network. Information security rules and responsibility.	Work in services, resources of the digital environment. Use the methods of preparing messages, searching, transferring information. Safely organize your personal information space.	Apply personal and collective, private and public digital communications services in life, creativity, study, work. To be included in new resources of networked smart systems. Adapt to AI devices and services.

derstand, explain information processes and digital objects of information activity), instrumental (be able to, own digital devices, resources and tools for working with them) and social (act in the digital world, independently apply in digital environment, transfer digital skills to new situations in the context of the emergence of innovative digital resources). The digital literacy framework is presented in the Table 1.

6. Educational Pillars for the Development of Algorithmic Thinking

Algorithmic thinking is always embedded in the context of digital literacy, constituting its intellectual and instrumental basis. The main value of algorithmic thinking in the digital literacy system is a change in the technical approach (training on specific information activity tools) to an algorithmic approach, that is, the formation of knowledge and skills to apply generalized algorithmic constructs in information activities in a digital environment, transferring them to new digital devices, tools and software environments independently.

The question arises, how to teach operational thinking based on an algorithmic approach? In school education, digital literacy is formed in the informatics course, where algorithmic thinking is the main thing, but transfers the acquired knowledge and skills to all school subjects. The difference in informatics is that it is a subject that, by shaping digital literacy, complements all the traditional types of children's literacy.

It can also be said that the intellectual pillars of algorithmic literacy are formed in mathematics and philological subjects such as mathematical and reading literacy based on algorithmic thinking in primary school and further in basic school in natural science and humanitarian subjects. This is done within the framework of research and modeling based on algorithmic thinking, including analysis and synthesis, concretization and generalization, logic and relationships, cause-and-effect relationships, systematization, etc.

The instrumental supports of algorithmic thinking are manifested in digital literacy and are formed in the technological and practical aspects of activities in natural science and humanitarian subjects, in information technology. They are also embedded in the objective activity of art and technology according to design algorithms, mastering the tools of creativity, and in health-saving subjects (physical culture, the basics of life safety), built on instructions and algorithms for behavior and decision-making.

7. Typical Educational Tasks Based on Algorithmic Constructs

The main problem of the formation of digital literacy of children on the basis of algorithmic thinking in the context of information activities is to offer a set of educational tasks that are built into all types of school subjects. These can be interdisciplinary algorithmic problems with obligatory computer support. Each task has an algorithmic step (intellectual construct) and a digital step (instrumental computer construct), which determine the educational activities of children to master digital literacy. From these steps, initial digital literacy is built for younger schoolchildren, which is important for their further development – this is a "digital bridge" to the new literacy of children.

Below are five content blocks of such typical tasks. They are intended for all age groups of schoolchildren, but they are implemented by different types of tasks, taking into account the age of the children. The result of mastering the blocks of educational tasks is the independent use of not only algorithmic approaches for information processing, but also tools for working on a computer, taking into account the age characteristics of children.

The highlighted blocks correspond to the groups of digital literacy competencies described above: information digital literacy – working with digital information, technical computer digital literacy and communication digital literacy. On the basis of algorithmic thinking and the practice of using digital devices and computer programs, intellectual (know, understand) and instrumental (be able, get digital experience) constructs of digital literacy are distinguished in each block.

7.1. *Block of Tasks "Models of Information Structures and Their Parameters"*

Intelligent constructs (algorithmic step): alphabet, transcription, musical notation, list, catalog, table, diagram (block diagram), graph, tree, network, diagram, graph, drawing, geometric shapes, text layout, formula (arithmetic expression), map, plan, pictogram, barcode, digital code, index, address, file, folder, directory.

Instrumental constructs (digital step): digital information and binary code; file system and screen interface constructs, digital text and its constructs (page, format, paragraph, fragment, font, hyperlink, table, illustration), configuring construct parameters; computer graphics constructs and their customization (raster, vector, graphic objects, object transformation, palette, color); multimedia constructs (video, sound on a computer, interactive environments); spreadsheet constructs, databases; constructs of e-mail, website, search engine.

7.2. *Block of Tasks "Digital Control Models: Algorithms and Programs"*

Intelligent constructs (algorithmic step): action plan, step, command; algorithmic structure (linear, branching, logic, cycle), algorithmic decision models; command writing rules, programming language syntax, choice from a set of commands, result analysis, command ordering plan logic.

Instrumental constructs (digital step): control of virtual and real executor of commands, system of executor's commands; computer – command executor, operating system, program interface tools, types of interfaces and interface devices (keyboard, remote control, voice control, touch panel, on-screen manipulator); application programs for processing data of various types, an algorithm of actions with tools of the software environment, menus, setting the parameters of tools in the program, computer template, layout, prototype, embedded objects.

7.3. *Block of Tasks "Models of Information Processes and Information Activities"*

Intelligent constructs (algorithmic step): actions – collecting and observing information, organizing information, fixing information (forms of presenting information of various types), processing information (methods of processing information of various types), storing information (directory and its structure, file, folder on devices storage, computer temporary memory and binary codes, cloud storage structure); form of information transfer (information structure of the digital communication system, data transfer), information search (search logic), information analysis (screen presentation of information, hypertext, augmented reality, information protection, information threats – an algorithm of actions).

Instrumental constructs (digital step): digital devices, digital device software, program menu interface, program tool setting interface, text input interface, digital writing, computer interface, browser command interface, search interface; means of information protection, rules of social behavior in the digital environment.

7.4. *Block of Tasks "Information Systems: Algorithmic Models in the Digital Environment"*

Intelligent constructs (algorithmic step): the world of automatic devices of software systems; algorithms of human functioning in the world of information systems, information systems for people with special needs; shadow internet and positive internet, MOOCs, e-learning platforms, cyber-physical systems, information systems and professions.

Instrumental constructs (digital step): digital steps – tests of digital competencies in a hybrid digital environment of people, cyber machines and robot programs and algorithms for their interaction; satellite navigation systems, examples of cyber-physical systems in the service of humans, unmanned devices, automated control systems; virtual reality and human, examples of cyberworlds, cyberculture and cyberart; systems with artificial intelligence, examples of smart systems in various fields of activity, in public space; digital life.

7. 5. *Block of Tasks "Models of Actions in the Environment of Digital Communications"*

Intelligent constructs (algorithmic step): algorithmic bases of interaction in computer networks; the procedure for safe actions in the digital environment, publicly available personal and collective options for digital interaction: video systems in society, network subcultures, the blogosphere, educational platforms, gaming platforms, e-commerce, social digital services; dealing with threats and countering dependence on the digital environment and the blogosphere.

Instrumental constructs (digital step): digital communication tools, digital governance – artificial intelligence, big data and society, social information systems, media content; social threats of the digital world – Internet content, services, big data in digital communications and algorithms for managing society through virtual space, bot programs, invasion of private and public space; psychological threats of the digital world in cyberspace – laykomaniya, Internet addiction, computer gambling addiction, digital crime; physical threats of the digital world and the variety of digital devices in everyday life – safety equipment, sanitary standards, physical inactivity and health preservation in the digital environment.

8. An Example of an Algorithmic Task Template for Younger Schoolchildren

The early development of algorithmic thinking is based on computer-based algorithmic problems. There are many examples of such tasks, but the activity aspect of development is most fully reflected in problematic tasks that stimulate the simplest approaches to modeling based on algorithms. Let us give an example of a template of algorithmic problems for younger schoolchildren from the set of microprojects "The Informatics Land" (Fig. 1), published in the Informatics problem book for younger **schoolchildren** (Tsvetkova, 2018; Tsvetkova, 2018).

The micro-project in each task is designed for one algorithmic step, the implementation of which is an independent achievement for the child, a discovery, like a eureka. This algorithmic step is the fulcrum for the development of new aspects of algorithmic actions and creativity based on these actions.

All tasks are built in the form of an algorithmic template "Robot Question" for seven thematic modules (Fig. 1): the Electronics River (computer structure), the Pier of Sets, the Pier of Logic, the Pier of Algorithms, the Pier of Patterns, the Pier of the coordinate Grid, and the Pier of Graphics. Computer support is provided by the open educational media resource "World of Informatics" in 4 parts (Cyril and Methodius, 2015), which children "attend" at the end of each lesson for 10–15 minutes according to the project

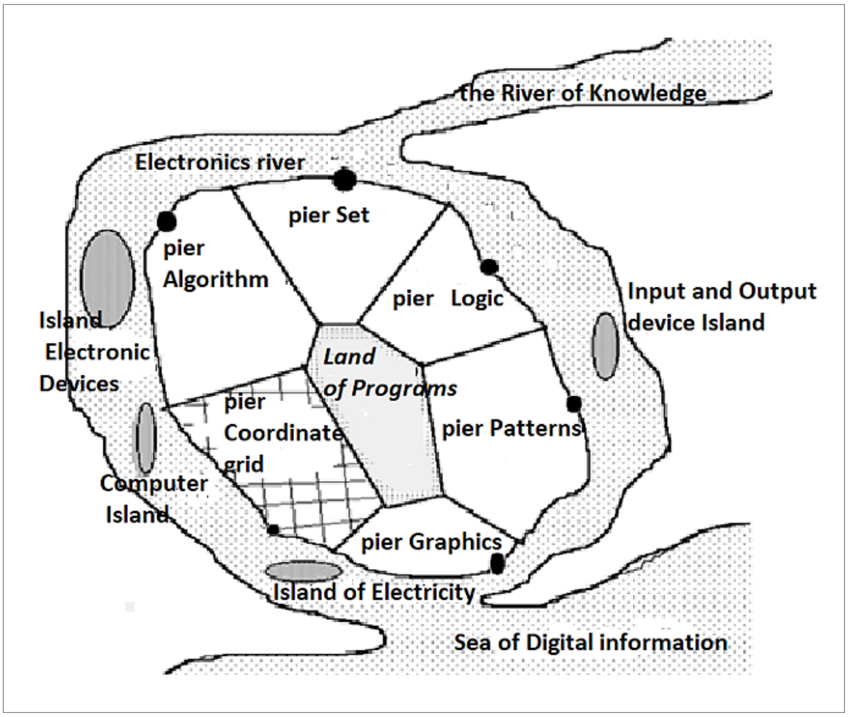


Fig. 1. Scheme of "Informatics Land".

scenario, and perform the tasks of programmed control of the robot Transporter. Additionally, a set of 100 algorithmic problems is presented in the book "Computer Science Virtual Laboratories" (Tsvetkova and Kuris, 2015): the concept of this algorithmic tasks is described in the article (Tsvetkova and Kiryukhin, 2016).

The microproject as an algorithmic task is intended to familiarize with the design activities of younger schoolchildren on the basis of algorithms. The goal is to teach a child to go through two stages in algorithmic activity: intellectual (to create a model for solving a problem and an algorithmic image) and instrumental (to draw up, show this solution by means of digital literacy on a computer as a computer sketch of an idea). As a result, children can complete a project in a subject environment using a computer or perform work in creative material based on a computer sketch with drawing tools, paper, textile design, assembly and control of a robot, a screen performer control program, mathematical actions according to an algorithm, or lexical actions (digital input , design of creative text, script), design of a physical training warm-up, observations, fixation and design in the form of illustrations, tables, diagrams, diagrams.

Thus, a computer sketch based on an algorithm becomes an educational support – **a digital step** from the idea of solving a problem to its embodiment in creativity, discovery.

The hero of all tasks is the robot Question, which sets the format of the computer sketch based on the algorithm diagram. The name of the robot, the virtual assistant to the schoolboy, was not chosen by chance. It is known that in case of difficulties when performing work on a computer, you can always use the "Reference" or "Help" command. Often this command is indicated by a question mark on the computer screen. Therefore, the hero of the problem book is called so – the robot Question.

In each topic of the practical work, he will be joined by new heroes in the form of robot- assistants. These are the Verbalist (will help us perform tasks for processing symbols: letters, numbers, signs), the Designer and the Builder (will help to construct various objects), the Artist and the Printer (will help us use drawing and graphics tools when doing the work), the Calculator (will help to solve problems, numerical puzzles and magic tricks), the Thinker and the Black Box (will help solve logical problems and find a plan-algorithm for solving problems), Traveler (will help to control the behavior of the executor in steps and directions), Postman (will help to find the necessary addresses), Scrambler (will help to complete tasks on message encoding and decoding).

To help in the solution, a template is used – a diagram of the algorithm for solving the problem. This scheme is the same for all tasks and is a drawing of a robot (Fig. 2). So, the "body" of the robot has three blocks for each task: input data, processing unit (algorithm) and output data (result). It is necessary to write down the task in the album, filling in the corresponding scheme – picture.

A micro-project scenario in any subject problem includes the ability to distinguish an algorithmic structure: input information (given), output information (solution), algorithm (model) of the solution. Hence, the levels of complexity (differentiation) of the microproject arise:

1) The input, the algorithm and the output are known (conceptual, demo).
2) The input and the model are known, build the exit (reproductive).

3) The algorithm and the output are known, construct the input (reverse reproductive).
4) The input and output are known, build an algorithm (productive).
5) the algorithm is known, pick up the input and output (reverse productive).

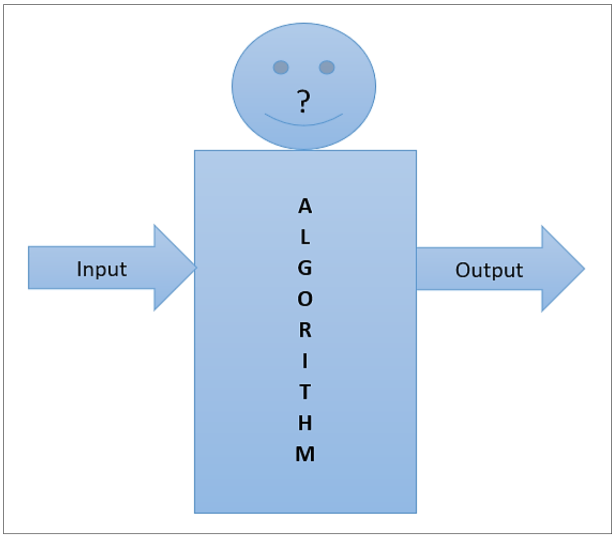


Fig.2. Problem diagram "Robot Question".

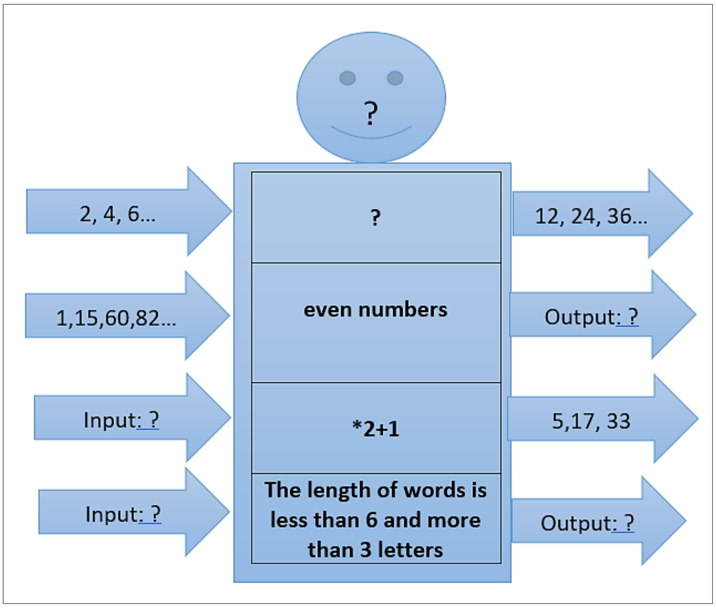


Fig. 3. Example task "Robot-Calculator".

Fig. 3 shows the robot Calculator task template that demonstrates the possible thresholds for the task's complexity.

Each section of the problem book offers a set of tasks on the topic of the section. The teacher and children can develop problem sets themselves by transferring algorithm templates for their ideas.

Conclusion

The development of algorithmic thinking in the system of mastering the basics of digital literacy by children from an early age is a new literacy in the digital transformation of the world. This knowledge and digital skills should be made universally for children, as should basic traditional literacy accessible to all children. It is a digital bridge to the ever-changing information world. It eliminates the digital divide for children and helps them make full use of the digital environment for learning and development, and find their professional mark in the near digital future.

In our time, the predictions of Academician A. Ershov (Ershov, 1981) remain relevant, that the task of the school is "to update the information model of the world in the form of programs". "The constant complication of the environment requires and at the same time makes it possible to significantly increase the intellectual power of mankind. A significant place in this progressive development of human intelligence should be occupied by the laws of information processing, ways of transition from knowledge to action, the ability to build programs, reason about them and foresee the results of their implementation".

The concept of new digital literacy based on algorithmic thinking, described in this article, is implemented in Russian primary schools in the form of an author's learning program, textbooks and manuals on Informatics (Tsvetkova *et al.*, 2019), as well as learning programs and textbooks on information security (Tsvetkova and Yakushina, 2020) for primary school.

Reference

Ershov, A. (1981). Programming, the second literacy. In: *Computer and Education: Proc. IFIP TC-3 3rd World Conf. on Computer Education – WCCE 81. Lousanne, Amsterdam, 1981. Part 1.* P.1–17 (in Russian: Программирование вторая грамотность. Выступление на 3-й Всемирной конференции ИФИП и ЮНЕСКО по применению ЭВМ в обучении, 27–31 июля 1981 г. в Лозанне (Швейцария): `http://ershov.iis.nsk.su/ru/second_literacy/article`

Ershov, A. (1988). School Informatics in the USSR. In: *Children in the Information Age: Opportunities for Creativity, Innovation and New Activities: Second International Conference – Sofia, Bulgaia, 19–23 May, 1987.* Pergamon Press, p. 37–57. `https://books.google.ru/books?id=ISaoBQAAQBAJ&pg=PA54&lpg=PA54&dq=WCCE+81.+Lousanne,+Amsterdam,+1981.+Part+1.+P.+1-17&source=bl&ots=iVE4vyBQMb&sig=ACfU3U1ToDGjStz0QTW95TeIONNqrOkcIA&hl=ru&sa=X&ved=2ahUKEwjSjs2JpLrwAhXltYsKHZQPB80Q6AEwCXoECAQQAw#v=onepage&q=WCCE%2081.%20Lousanne%2C%20Amsterdam%2C%201981.%20Part%201.%20P.%201-17&f=false`

Papert, S. (1988). A critique of technocentrism in thinking about the school of the future. In: *Children in the Information Age: Opportunities for Creativity, Innovation and New Activities: Second International Conference – Sofia, Bulgaia, 19–23 May, 1987*. Pergamon Press, p. 3–19. https://books.google.ru/books?id=ISaoBQAAQBAJ&pg=PA54&lpg=PA54&dq=WCCE+81.+Lousanne,+Amsterdam,+1981.+Part+1.+P.+1-17&source=bl&ots=iVE4vyBQMb&sig=ACfU3U1ToDGjStz0QTW95TeIONNqrOkcIA&hl=ru&sa=X&ved=2ahUKEwjSjs2JpLrwAhXltYsKHZQPB80Q6AEwCXoECAQQAw#v=onepage&q=WCCE%2081.%20Lousanne%2C%20Amsterdam%2C%201981.%20Part%201.%20P.%201-17&f=false

Piaget, J. (1985). *Equilibration of cognitive structures*. University of Chicago Press. https://piaget.org/

Vygotsky, L. (1934). *Thinking and Speaking* (in Russian: Мышление и речь. М.; Л.: Соцэкгиз, 1934. P.323)

Vygotsky, L., Luria, A. (1930). *Tool and Symbol in Child Development*. https://www.marxists.org/archive/vygotsky/works/1934/tool-symbol.htm

Blonsky, P. (1935). *Memory and Thinking* (in Russian: Память и мышление. Москва, 1935, с. 215): http://elib.gnpbu.ru/text/blonsky_pamyat-myshlenie_1935/go,0;fs,1/

Rubinshtein, S. (1989). *Fundamentals of General Psychology* (in Russian: Основы общей психологии. В 2-х томах, Т. 2, Часть 4. М, Педагогика, 1989): http://elib.gnpbu.ru/text/rubinshteyn_osnovy-obschey-psihologii_t2_1989/go,5;fs,1/

Zankov, L. (1975). *Learning and Development* (Experimental Pedagogical Research) (in Russian: Обучение и развитие (Экспериментально-педагогическое исследование). М.: Педагогика, 1975. 440 с.)

Elkonin, D. (2007). *Child Psychology: Tutorial*. Publishing Center "Academy", Moscow (in Russian: Детская психология: учебное пособие. М.: Издательский центр «Академия», 2007. 384 с): http://psychlib.ru/mgppu/Edp-2007/Edp-001.htm#$p1

Davydov, V. (1996). *Developmental Learning Theory*. Moscow, INTOR (in Russian: Теория развивающего обучения. М.: ИНТОР, 1996): http://www.psy.msu.ru/people/davydov.html

Davydov, V. (1995). *On the Concept of Developing Education: Collection of Articles*. "Peleng", Tomsk (in Russian: О понятии развивающего обучения: сб. статей. «Пеленг», – Томск, 1995): http://elib.gnpbu.ru/text/davydov_o-ponyatii-razvivayuschego-obucheniya_1995/go,0;fs,0/

Galperin, P. (1985). *Teaching Methods and Mental Development of the Child*. (in Russian: Методы обучения и умственное развитие ребенка. – М., 1985): http://www.psy.msu.ru/people/galperin.html

Leontiev, A. (1975). *Activity. Consciousness. Personality*. (in Russian: Деятельность. Сознание. Личность. – М, 1975): http://www.psy.msu.ru/people/leontiev/dsl/index.html

Shapiro, S. (1975). From algorithm to judgment. (in Russian: От алгоритма к суждению. М, Советское радио, 1975, с. 288)

Tsvetkova, M. (2018). *Informatics. Problem book for grade 3*. (in Russian: Информатика. Задачник для 3 класса. Издательство "БИНОМ". М. 2018): https://lbz.ru/books/750/7762/

Tsvetkova, M.S. (2018). *Informatics. Problem book for grade 4*. (in Russian: Информатика. Задачник для 4 класса. Издательство "БИНОМ". М. 2018): https://lbz.ru/books/750/9159/

Cyril and Methodius (2015). World of Informatics. Open electronic resource (OER), Cyril and Methodius Company. (in Russian: Мир Информатики. Открытый электронный ресурс в 4-х ч., "Кирилл и Мефодий" – М. 2015): https://lbz.ru/metodist/authors/informatika/5/ep-4-umk3-4fgos.php

Tsvetkova, M., Kuris, G. (2015). Virtual laboratories for Informatics in primary school: a methodological manual and Open electronic resource (OER). (in Russian: Виртуальные лаборатории по информатике в начальной школе: методическое пособие и открытй электронный ресурс. Издательство "БИНОМ". М. 2015): https://lbz.ru/books/1112/5211/

Tsvetkova, M., Kiryukhin, V. (2016). Concept of Algorithmic Problems for Younger Students Olympiads in Informatics. *Olympiads in Informatics*, 10 (special issue), 67–78.

Tsvetkova, M., *et al.*, (2019). Educational-methodical set of the team of authors "Informatics" 3–4 classes. (In Russian: УМК авторского коллектива «Информатика» 3–4 классы. Издательство "БИНОМ". М.): https://lbz.ru/books/750/

Tsvetkova, M., Yakushina, E. (2020). Information Security. Safe Internet Rules. Grades 2–4: tutorial. (In Russian: Информационная безопасность. Правила безопасного Интернета. 2–4 классы: учебное пособие. Издательство "БИНОМ". М.): https://lbz.ru/books/1097/11206/

M.S. Tsvetkova, professor of the Russian Academy of Natural Sciences, PhD in pedagogic science, prize-winner of competition “The Teacher of Year of Moscow” (1998). from 2002 to 2018 she is a member of the Central methodical commission of the Russian Olympiad in informatics and the pedagogic coach of the Russian team on the IOI. She is the author of many papers and books in Russia on the informatization of education and methods of development of talented students. She is the author of official textbooks and copybooks in Russia for primary school in Informatics. She is author and director of the International school in Informatic ISIJ (since 2017). She is the Russian team leader (2013–2017). She was awarded the President of Russia Gratitude for the success organizing the training of IOI medalists (2016). She is now the Expert of Committee on Education and Science State Duma of the Russian Federation (since 2017), and she has the Committee on Education and Science State Duma Gratitude (2021).

V.M. Kiryukhin is professor of the Russian Academy of Natural Sciences. He is the author of many papers and books in Russia on development of Olympiad movements in informatics and preparations for the Olympiads in informatics. He is the exclusive representative who took part at all IOI from 1989 to 2017 as a member of the IOI International Committee (1989–1992, 1999–2002, 2013–2017) and as the Russian team leader (1989, 1993–1998, 2003–2012). He received the IOI Distinguished Service Award at IOI 2003, the IOI Distinguished Service Award at IOI 2008 as one of the founders of the IOI making his long term distinguished service to the IOI from 1989 to 2008 and the medal “20 Years since the First International Olympiad in Informatics” at the IOI 2009. He was the chairman of the IOI 2016 in Russia, and has the award medal of the President of Russia (2016) for organizing the Olympiad in Informatics in Russia and training IOI medalists since 1989. He is now the President of the international organizing Committee ISIJ.

Olympiads in Informatics, 2021, Vol. 15, 119–132
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.10

Look Ma, Backtracking without Recursion

Dedicated to my colleague Ruurd Kuiper, on the occasion of his retirement

Tom VERHOEFF
Mathematics and Computer Science, Eindhoven University of Technology
Groene Loper 5, 5612 AE, Eindhoven, Netherlands
e-mail: t.verhoeff@tue.nl

Abstract. I show how backtracking can be discovered naturally without using a recursive function (nor using a loop with an explicit stack). Rather, my approach involves a form of self application that can be elegantly expressed in an object-oriented program, and that is reminiscent of how recursion is done in lambda calculus. It also illustrates why reasoning about object-oriented programs can be hard.

Keywords: computer science, programming, object-oriented, functional, backtracking, recursion, fixed-point combinator, self application.

1. Introduction

This article can be viewed as an addition to Verhoeff (2018), which covers – what I believe to be – the basics of recursion. I will illustrate my approach to backtracking through the problem of solving simplified Binairo puzzles, explained below. Section 2 considers various reasoning strategies to solve such puzzles, culminating in a non-recursive backtracking strategy. Design details, Java code, and some refinements are discussed in Section 3, and the essence is extracted in Section 4. Section 5 concludes the article. I recommend that younger programmers first read §2 and §3.2 and then explore the source code provided in Verhoeff (2021), before reading the other sections.

2. Reasoning Strategies to Solve Puzzles

The puzzles in this article are simplified *Binairos*[1], consisting of a square grid with an even number of rows and columns, partly filled with zeroes and ones (see Fig. 1). The objective is to fill the grid completely with zeroes and ones such that (**Rule 1**) *nowhere three*

[1] A.k.a. binary or Takuzu puzzles; see `https://en.wikipedia.org/wiki/Takuzu`

0		**0**		**1**			
					1	**1**	
0	**0**			**1**			
0						**1**	**0**
			1				
				1			
			1				**0**

Fig. 1. Initial state of our 8 × 8 example puzzle.

or more equal symbols are horizontally or vertically adjacent, and (**Rule 2**) *in each row and in each column, the number of zeroes equals the number of ones*. The simplification is that we allow identical rows and columns. The given zeroes and ones cannot be changed when solving the puzzle, and you may assume that there is a unique solution. Try to solve the example in Fig. 1 if you have not done this kind of puzzle before.

Puzzles can often be solved just by reasoning, and there is no need for 'blind' backtracking. Given Rules 1 and 2 for our puzzles, there are two obvious strategies to fill in what one could call *forced* bits.

1. If in three horizontally or vertically adjacent cells two cells have the same bit b, then the other cell must have the opposite of b, i.e. $1 - b$.
2. If in a row or column half of its cells have bit b, then the remaining cells must have $1 - b$.

The correctness of these strategies follows from the fact that a solution exists. We call three horizontally or vertically adjacent cells a *triplet*, and a complete row or column a *line*. In pseudocode, these strategies can be expressed as follows.

1. *Triplet* strategy: Find a triplet t with one empty cell and a bit b, where b occurs twice in t, and fill the empty cell with bit $1 - b$.
2. *Line* strategy: Find a line ℓ with at least one empty cell and a bit b, where b occurs in half of ℓ's cells, and fill all empty cells of ℓ with bit $1 - b$.

Note that such strategies take a puzzle as argument, and return a, possibly updated, puzzle. We will leave that puzzle parameter implicit for now (think of it as a global variable; see the next section for details). These strategies are nondeterministic, and when they do not apply, they leave the puzzle unchanged. *Strategies must have the property that* (*i*) *they only change empty cells, and* (*ii*) *they preserve solvability*, i.e., if the puzzle was solvable *before* applying the strategy then it is still solvable *after* applying the strategy.

If you have applied a strategy and nothing changed, then it does not make sense to apply it again. But if it did bring some change, then it might be applicable again. Moreover, if the triplet strategy did not bring any changes, but subsequent application of the line strategy did, then it would be good to try the triplet strategy again (see Fig. 2).

This leads to the wish to define strategy *combinators* such as the following.

3. *Fixed-point* strategy: Repeatedly apply a given strategy until no further change occurs. It takes a strategy as parameter.

0	*1*	**0**		**1**			
1				*0*	**1**	**1**	*0*
0	**0**	*1*	*0*	**1**		*0*	*1*
0						**1**	**0**
1			**1**				
			0	**1**			
			1				**0**

0	*1*	**0**		**1**			
					0		
1				*0*	**1**	**1**	*0*
0	**0**	*1*	*0*	**1**	$\underline{1}$	*0*	*1*
0					*0*	**1**	**0**
1			**1**				
			0	**1**			
			1				**0**

Fig. 2. State after applying the *FT* strategy to Fig. 1 (left), and after the *FPFTL* strategy (right), where the line strategy yielded the blue (underlined) 1.

4. *Pair* strategy: Apply two given strategies one after the other. It has two strategy parameters, and can be nested to combine more strategies.

We call these *meta-stategies*, because they take one or more strategies as parameter. If we (as humans) consider the triplet strategy to be simpler to apply than the line strategy, then we would probably use the following strategy *FPFTL*.

5. *FPFTL* = *fixed_point* (*pair* (*FT*, *line*)) where
6. *FT* = *fixed_point* (*triplet*).

Applying the *FT* and *FPFTL* strategies to the puzzle in Fig. 1 leads to Fig. 2.

For the example puzzle, we are then stuck. You may already have discovered the following strategy to help out.

7. *Contradiction* strategy: Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state *after applying the FPFTL strategy*, and fill cell c with bit $1 - b$.

The contradiction strategy speculates and looks ahead. It is correct, because by assumption there exists a solution. You can see it in action in Fig. 3. In fact, *fixed_point* (*pair* (*FPFTL*, *contradiction*)) solves our example puzzle.

0	*1*	**0**		**1**			
$\tilde{0}$					*0*		
1				*0*	**1**	**1**	*0*
0	**0**	*1*	*0*	**1**	*1*	*0*	*1*
0					*0*	**1**	**0**
1			**1**				
1			*0*	**1**			
1			**1**				**0**

0	*1*	**0**	$\underline{0}$	**1**	*1*	*0*	*1*
$\tilde{1}$	*1*	*0*	*1*	*0*	*0*	*1*	$\underline{0}$
1	$\tilde{0}$	*1*	$\underline{0}$	*0*	**1**	**1**	*0*
0	**0**	*1*	*0*	**1**	*1*	*0*	*1*
0	*1*	*0*	*1*	$\underline{1}$	*0*	**1**	**0**
1	$\underline{0}$	$\tilde{0}$	**1**	*0*	*1*	*0*	*1*
0	*1*	*1*	*0*	**1**	$\underline{0}$	*0*	*1*
$\underline{1}$	*0*	$\underline{1}$	**1**	*0*	$\tilde{0}$	*1*	**0**

Fig. 3. When trying 0 in the red cell (with tilde), and applying *FPFTL*, the line strategy yields the 1s in the yellow cells, violating Rule 1 (left); therefore the red cell must contain 1; applying *fixed_point*(*pair*(*FPFTL, contradiction*)), we get the shown solution (right), where the colors encode the responsible strategy: yellow–triplet, blue(underline)–line, red(tilde)–contradiction.

Observe that the pair of triplet and line strategies together can be viewed as special case of the following direct contradiction strategy.

8. *Direct contradiction* strategy: Find an empty cell c and bit b, where putting bit b in cell c directly leads to an invalid state, and fill cell c with $1 - b$.

Note that this may be less efficient, but we now do have

9. *FPFTL* = *fixed_point* (*direct_contradiction*).

To avoid code duplication (DRY = Don't Repeat Yourself), let's see if we can unify the code for these two contradiction strategies by generalization via a strategy parameter, making it a meta-strategy.

10. *General contradiction* strategy: Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state *after applying a given strategy*, and then fill cell c with $1 - b$.

Its usefulness depends on how good the supplied strategy is at finding forced bits. But even when the supplied strategy does nothing, the general contradiction strategy is correct. Let's define the *empty* strategy as doing nothing (applying the identity function).

11. *Empty* strategy: Do nothing.

Then we see that both contradiction and direct contradiction are indeed special cases of general contradiction.

12. *contradiction* = *general_contradiction* (*FPFTL*).
13. *direct_contradiction* = *general_contradiction* (*empty*).

In fact, now we no longer need *FPFTL*, because it can be expressed in terms of the general contradiction strategy via 9 and 13, and the contradiction strategy now becomes a double application of general contradiction:

14. *FPFTL* = *fixed_point* (*general_contradiction* (*empty*)).
15. *contradiction* = *general_contradiction* (*fixed_point* (*general_contradiction* (*empty*))).

2.1. *Self Application*

Then the idea occurred to me that instead of *FPFTL* in the contradiction strategy, we should use the best strategy we can think of to find a sequence of forced bits leading to a contradiction. So, what about supplying itself as parameter? Like this:

16. *general_contradiction* (*general_contradiction* (*general_contradiction* (...))).

Of course, we cannot define it with those dots. But we can define a variant H of the general contradiction strategy that expects a hyper-strategy h as parameter, viz. a strategy that takes a hyper-strategy (not necessarily itself) as parameter.

17. *Hyper-contradiction* (*hyper-strategy* h): Find an empty cell c and a bit b, where putting bit b in cell c leads to an invalid state after applying strategy $h(h)$, and fill cell c with $1 - b$. N.B. h could ignore its argument!

Therefore, the hyper-contradiction strategy is in fact also a hyper-strategy, and can be passed as parameter to itself. Now we can properly define the strategy in 16 as $H(H)$. Note that this is a regular strategy (not meta or hyper).

It works (thanks to the two key properties of strategies), but it is not guaranteed to solve puzzles by itself. It may have to be repeated, which we can do by applying the fixed-point strategy. But if that is a better strategy, then we want to use that as parameter in the hyper-contradiction strategy. This can be accomplished by hyper-strategy *FH* defined by

18. $FH\,(hyper\_strategy) = fixed\_point\,(hyper\_contradiction\,(hyper\_strategy))))$

and applying it to itself, which basically gives us backtracking!

19. $backtracking = FH(FH)$.

If there exists a *unique* solution (which is assumed), anything other than the correct bit in a cell must lead to a contradiction. And it terminates because strategies only fill empty cells. Note that if the puzzle has multiple solutions, then our *backtracking* strategy won't find any of them, because then there exists at least one cell where both 0 and 1 are valid, and there won't be a contradiction.

Observe that these function definitions are not recursive (though there is self application). Of course, there are some details to take care of to make all of this work in a real (strongly typed) programming language. For that, see the next section.

3. Design Details, Java Code, and Refinements

One could try to implement my approach to backtracking in a functional programming language, as sketched above. But this can lead to a typing problem because of the self application. I found it a nice challenge to code it in the object-oriented language Java (which we use in our education, and which used to be permitted at the International Olympiad in Informatics). To keep the code easily understandable for non-Java programmers, I stick to a minimal subset, avoiding interfaces and generic type parameters. Also see (Verhoeff, 2018, §6) for how to type and define functions that can be self-applied in Java.

First, I look at some design details in §3.1, still using a functional approach. Note that these are mathematical functions, and not programming-language functions, that is, without side effects operating on (immutable) values. Next, I present an overview of the Java source code, and finally, I discuss some refinements (that could serve as exercises).

3.1. *Design Details*

Here, I will elaborate on some of my design decisions to obtain an object-oriented (OO) program from the functional description in Section 2. The focus is on implementing strategies. Recall that a strategy is a function from the domain of puzzles, denoted by $\mathcal{P}$, to itself, i.e., they have type $\mathcal{S} = \mathcal{P} \to \mathcal{P}$. In the functional setting there is no mutable data, only values. So, in OO terms, a strategy returns a fresh object, copied from its argument and possibly altered. In the OO world, this is frowned upon, because the updates to the puzzle are quite local. So, a mutable type of puzzles is preferred, to avoid copying lots of data that is unchanged. (In functional programming this is frowned upon, and one would use lazy updaters applied to the puzzle, but that is outside the scope of this article. See the Python code in Verhoeff (2021) for the idea.)

Before dealing with the puzzle parameter of a strategy, let's also look at meta-strategies. These have type $\mathcal{M} = \mathcal{S} \to \mathcal{S}$. In a functional setting, this is a *curried* function of two arguments, viz. first a strategy and then a puzzle yielding a puzzle: $\mathcal{M} = \mathcal{S} \to \mathcal{P} \to \mathcal{P}$, where the latter is to be read parenthesized as $\mathcal{S} \to (\mathcal{P} \to \mathcal{P})$. Thus, if $m \in \mathcal{M}$, $s \in \mathcal{S}$, and $p \in \mathcal{P}$, we have $m(s) \in \mathcal{S}$ and $m(s)(p) \in \mathcal{P}$. *Uncurried*, the latter would be written as $m(s, p)$. The call $m(s)$ is called a *partial application* of m, because it lacks the second argument, which is needed to start the evaluation.

Let's introduce shorter names for the various strategies introduced in §2.

- $T \in \mathcal{S}$ – triplet strategy (see §2).
- $L \in \mathcal{S}$ – line strategy (see §2).
- $F \in \mathcal{M}$ – fixed-point meta-strategy, satisfying

$F(s)(p) = p$ **if** $s(p) = p$ **else** $F(s)(s(p))$.

This is recursive, but in a Java program, this would be done via a `do-while` loop, not needing a stack. It terminates because every update that strategy s makes to p changes empty cells only.

- $P \in (\mathcal{S} \times \mathcal{S}) \to \mathcal{S}$ – pair meta-strategy, taking a pair of strategies as argument, such that $P(s_1, s_2)(p) = s_2(s_1(p))$; this corresponds to function composition: $P(s_1, s_2) = s_2 \circ s_1$.
- $E \in \mathcal{S}$ – empty strategy, with $E(p) = p$ for all $p \in \mathcal{P}$.
- $G \in \mathcal{M}$ – general contradiction meta-strategy (see §2).
- $H \in \mathcal{H} \to \mathcal{S}$ – hyper-contradiction hyper-strategy, where we have $\mathcal{H} = \mathcal{H} \to \mathcal{S}$ (this is an infinite type that most functional languages don't like, but that we can get to work in Java).

Unfortunately, in OO programming, currying and partial application don't come for free. Suppose we have a function $f \in A \times B \to C$ with curried version $f' \in A \to B \to C$. In that case, $f'(a) \in B \to C$ is a partial application of f, and $f'(a)(b) = f(a, b)$. How can this be done in Java, where there are no separate functions? Instead, we have a method m – static or not – in some class C:

```java
class C {
    int m(int x, int y) {
        return x * x + y;
    }
}
```

To use this function, create an instance of its class, and call the method:

```
    C obj = new C();
    System.out.println(obj.m(1, 3)); // 4
    System.out.println(obj.m(2, 3)); // 7
```

To define a function partially applied only to its first argument, define a new class as carrier for such partially applied functions:

```
class PartialCm {
    private C obj; // 'receiver'
    private int x; // first argument

    PartialCm(C obj, int x) {
        this.obj = obj;
        this.x = x;
        // don't call obj.m, but store x ('lazy')
    }

    int apply(int y) {
        return obj.m(x, y); // call with both arguments
    }
}
```

Finally, we create objects from this class to get partially applied versions of m:

```
    // create partial applications
    PartialCm m_1 = new PartialCm(obj, 1);
    PartialCm m_2 = new PartialCm(obj, 2);
    // apply them
    System.out.println(m_1.apply(3)); // 4
    System.out.println(m_2.apply(3)); // 7
```

You could consider this an OO *design pattern* for partial function application. Unfortunately, it is quite bureaucratic and verbose for such a simple idea. You can see a resemblance to the Command design pattern here. That pattern is typically used to capture *all* parameters, and just delay the call. The only reason to delay a call in an OO language is because there is a side effect that must be properly synchronized with other actions. In a functional language, there are no side effects in function calls, and because of lazy ('on-demand') evaluation, you just call the function right away, relying on the compiler and runtime system to decide whether its execution is really needed.

When defining a function in an OO language, client code can pass the arguments to a method in several ways:

- Via parameters of the method (at time of the call).
- Via instance variables in the method's object (in advance of the call), with these variants:

  - via one or more, possibly parameterized, *constructors* (only once);
  - via one or more, possibly parameterized, *setter methods*;
  - via *direct access* to the (non-private) instance variables.

We need to decide how to handle the parameters of (meta-)strategies.

We opt for a puzzle class with mutable objects, and all the strategies will work on the same object. Because of mutability we now also need to worry about reverting changes made to the puzzle state in contradiction strategies. We choose to pass the puzzle parameter via a **`static`** instance variable, set once by the client code. That way we also avoid the need for defining constructors in subclasses that are not meta-strategies.

Concerning strategy parameter(s) of meta-strategies, we did the following. Since we cannot pass pure functions, we pass an object which the intended function as instance method. Also see Verhoeff (2012) on passing functions as arguments in Java.

- For the fixed-point and pair strategies, we pass them via their constructor and store them in instance variables. Polymorphism allows these strategies to abstract from the precise nature of their strategy parameters.
- For the contradiction strategies, we use instance variables set directly by the client code. In Section 4, you can see how this could have been avoided.

3.2. *Java Code*

The Java source code demonstrating that the approach to backtracking sketched in Section 2 really works is available in Verhoeff (2021). I have attempted to keep the code as simple as possible. To do so, I have sacrificed some good OO programming habits. For instance, I have omitted access modifiers (**`public`** and **`private`**) wherever possible. This makes instance variables non-encapsulated, and hence we don't need setters and getters.

First, an overview of the classes, focusing only on the essentials.

- `Puzzle` – a mutable puzzle with a square grid of `Cells`; all cells are also available for easy traversal in a single `Group` (for the contradiction strategies), and via lists of all `Triplets` and all `Lines` (for the corresponding strategies); boolean methods `isValid` (to check for rule violations) and `isSolved`.
- `Cell` – a mutable cell with its state, and some global constants.
- `Group` – a list of `Cells`; generalizes `Triplet` and `Line`, i.e., it has common code for
  - constructing a group from a rectangular block in a puzzle's grid
  - frequency counting of cell states (for validity check and for strategies)
  - bulk filling of empty cells (for triplet and line strategies)

  Subclasses:

  - `Triplet` and `Line` – implement `isValid` (for `Puzzle.isValid` and the contradiction strategies)
- `Strategy` – abstract base class for strategies; carrier of the method `apply` that applies the strategy to the **`static`** `puzzle` injected by the client; `apply` must be

an instance method to allow meta-strategies, such as the pair strategy, to operate on arbitrary strategies; see below for parameters and returned value of `apply`.

Subclasses:

- ◦ `TripletStrategy` and `LineStrategy` – to fill in a forced bit based on the rules for validity of `Triplet` and `Line`
- ◦ `FixedPointStrategy` – to repeat given strategy until no change; the strategy to repeat is injected via the constructor
- ◦ `PairStrategy` – to apply two strategies after each other; these strategies are injected via the constructor
- ◦ Not present: `ContradictionStrategy` – this is done in `Tests` via `GeneralContradictionStrategy`
- ◦ `EmptyStrategy` – to do nothing
- ◦ `DirectContradictionStrategy` – special case of the general contradiction strategy using the empty strategy as helper; no longer needed
- ◦ `GeneralContradictionStrategy` – to look for a contradiction after applying a given strategy; the strategy to help find a contradiction is set directly by the client code; also used to define the hyper-contradiction hyper-strategy
- ◦ Not present: `HyperContradictionStrategy` – is already offered by `GeneralContradictionStrategy`, because the strategy parameter `strat` passed as instance variable, can be a hyper-strategy (which also has type `Strategy`); self application cannot work via the constructor; client code sets it directly; this needs to be done only once, since all applications of the hyper-contradiction strategy will have the same actual strategy parameter (viz. itself)

- `Command` – abstract base class for commands that modify the puzzle's state; supports reverting an executed command; uses the Command design pattern.

  Subclasses:

  - ◦ `SetStateCommand` – command to set and revert state of given cell
  - ◦ `CompoundCommand` – a list of commands executed in sequence; uses the Composite design pattern; in traditional recursive backtracking, the old cell state is stored in a local variable of the recursive invocation instead of a command

- `Logging` – a utility class with **`static`** methods to do logging.
- `Tests` – class with `main` method to run some tests.
- `LookMa` – class with `main` method to run and time self-applied strategies.

There is one more thing worth discussing here, and that is how the method `Strategy.apply()` evolved as we added strategies.

1. **`void`** `apply()` – suffices for triplet, line, pair, and empty strategies.
2. **`boolean`** `apply()` returning whether puzzle was changed – needed for fixed-point strategy (to avoid copying the old state and comparing it); other strategies ignore the result.

3. **`int`** `apply()` returning number of changes – useful for logging; fixed-point strategy compares result to 0; other strategies ignore the result.
4. `Command apply()` returning all applied puzzle changes, that can be reverted; needed for (general) contradiction strategy; the fixed-point strategy uses the command's size; other strategies ignore the result .
5. `Command apply(`**`int`**` level)` – useful to restrict self-nesting depth and to do indented logging; this is in the current code base.
6. instance variable `untilFirstChange` – (default value **`true`**) added later to control whether strategies stop at the first change (in an earlier version, strategies would complete one sweep, possibly accumulating multiple changes; that behavior can be obtained by setting `untilFirstChange` to **`false`**); the current behavior is useful when you want to give a single hint; repeated application must now be obtained through the fixed-point strategy.

3.3. *Refinements*

There are many ways in which this approach can be improved. I present some suggestions as exercises.

- The pair strategy can be generalized to the *compound* strategy, which operates on a list of strategies. Implement it using the Composite design pattern, where one can dynamically add strategies at run-time to an initially empty strategy. The empty compound strategy can now be used instead of the empty strategy.
- The triplet/line strategies, iterate over the triplets/lines and then over that group to fill empty cells as applicable. Instead, iterate over the grid's empty cells, and for each cell iterate over its triplets/lines, to fill the empty cells as applicable. For this, each cell needs to know in what triplets and lines it occurs. (This also turns out to be useful for other refinements.)
- Typically, one uses only the line strategy (instead of *FPFTL*) in the contradiction strategy. In fact, contradiction often only applies the line strategy to the two lines that intersect at the empty cell being considered. For that, it would help if each cell would know in what lines it occurs.
- The presented backtracking strategy performs badly. If it tries bit 0 and this is correct, then it will find a solution, which it ignores because there is no contradiction. Then it will try bit 1, which of course leads to a contradiction (because there is only one solution). And only then will it conclude that there must be a 0 in the tried cell. When the example puzzle in Fig. 1 is solved by the self-applied fixed-point hyper-contradiction hyper-strategy, it finds 33 554 431 solutions along the way.
- We can remedy this by reporting solutions early, e.g., by throwing an exception in `SetStateCommand`. Then we have a more traditional backtracking algorithm. It would also help to report contradictions early, again by throwing an(other) exception.
- Improve performance further, by maintaining an instance variable in `Group` with the frequency counts of the group's cell states, to avoid their repeated re-compu-

tation. For that, each cell needs to know in which groups it occurs, so that when it changes state, all (and only) the relevant counts can be updated. Instead of the counts, you could maintain a list of cells per state, to make it easy to traverse cells of a particular state in that group.

- To simplify experimentation with strategies, define a Domain-Specific Language (DSL) to construct strategies. For example, in the functional notation of §3.1, the strategy expression $F(P(F(P(F(T), L))\ C))$ is not so complicated, but in the Java code, it becomes a multi-statement code fragment. I would prefer to write an even shorter expression like $((T^*; L)^*; C)^*$ and have this interpreted or expanded automatically.

4. Object-Oriented Programming is Hard

Object-oriented programming (OOP) offers various powerful language features, but these need to be used with care. In this section, I will boil down my approach to the bare essence, thereby pinpointing one of the pains of OOP.

Edsger Dijkstra (1968) fulminated against the *goto* statement, because it made reasoning about (the correctness of) programs unnecessarily hard (look up: ‘spaghetti code’). *Structured programming* did away with the goto statement, by restricting the flow of control to language constructs with unique entry and exit points (e.g., **`if-else`**, **`for`**, **`while`**). A next step in the evolution of programming languages incorporated *procedural abstraction*, where one can introduce named parameterized abbreviations for groups of statements, a.k.a. *functions*. And then came *data abstraction*, where one can introduce named parameterized (generic) type abbreviations for groups of variables and related operations, a.k.a. *classes*. Programming with such classes is often referred to as object-oriented programming. Dijkstra (1989) considered OOP “an exceptionally bad idea which could only have originated in California”. Here is an example to show why. Consider the following class.

```java
class Selfish {
    void f(Selfish x) {
          System.out.println("Working ...");
          x.f(x);
          // Not iterative/recursive, but self-applicative
          System.out.println("... and done!");
    }
}
```

Let me talk you through the code, and in the meantime you can try to see the connection to backtracking via self application. The class `Selfish` has no instance variables and a single instance method `f`, that takes an object of type `Selfish` as parameter. Note that the Java compiler compiles this, even though it is a cyclic type definition. Moreover, note that the *compile-time* type of parameter `x` is `Selfish`. Thanks to the support of *polymorphism*, however, the actual *run-time* type of the argument supplied to `f` can be any *subclass* of `Selfish`. Therefore, it is unclear (at compile-time) which `f`-functionality is invoked on line 31. This is worse than a goto statement, because with a goto statement, its control destination is known at compile-time. But with polymorphism, the control destination depends on the run-time circumstances (and could even depend on external input).

Here is a subclass of `Selfish`:

```java
class Innocent extends Selfish {
    @Override
    void f(Selfish x) {
            System.out.println("I'm innocent!");
    }
}
```

Class `Innocent` overrides the behavior of `f`. Now consider the following objects and calls of `f` (see `DemoSelfish.java` in Verhoeff (2021)). Can you predict the execution result?

```java
    Selfish omega = new Selfish();
    Selfish innocent = new Innocent();

    innocent.f(innocent);
    innocent.f(omega);
    omega.f(innocent);
    omega.f(omega);
```

No loop, no (static) recursion, but still a disaster happens. And that in such a small piece of code. Thanks to dynamic (re)configuration. Spaghetti code is bad, but goto statements are at least static. This example demonstrates something far worse: dynamic spaghetti. It makes for a great job interview question. Here, I rest my case.

5. Conclusion

The title of this article refers to the meme "Look ma, no hands"[2], said by kids proudly showing off to their mom that they can ride a bike without hands on the handlebars (it is also deployed in various jokes). That also captured my feeling when I discovered the approach to backtracking presented here.

I demonstrated that the power of self application can be discovered quite naturally in the context of what is traditionally called backtracking, e.g. when solving combinatorial puzzles. It gives rise to programs that do not employ *static* recursion,

[2] `https://wordhistories.net/2020/04/14/look-no-hands/`

where the body of function *f* contains function calls that can be traced statically (i.e., at compile time) to a call of *f* itself, but rather that employ a form of *dynamic* recursion. In the latter form, the traditional recursive calls are generalized (abstracted) to an additional function parameter, say *g*, of *f* (in OO, *g* will be a method of a parameter object). So, when one reasons about the program text of *f*, one does not know what the actual parameter *g* will be. Client code can later decide what function to provide for *g* in the call of *f*. You obtain recursion when calling *f* with itself as actual parameter for *g*. It is now also clear that one needs to reason about such programs in terms of *contracts*, that specify the pre-and post-conditions (assumptions and effects) of the functions *f* and *g*. Contractual reasoning is also the only way to get to grips with dynamic spaghetti.

To avoid misunderstandings, I mention two caveats.

- I am not claiming that the approach to backtracking in this article is to be preferred. It is purely meant as an interesting and possibly insightful approach. In fact, the version that I presented is inefficient, though it can be made efficient.
- The phrase 'without recursion' in the title is open to debate. But it is certainly not recursion in the traditional sense: a function having static calls to itself, or a container class having instance variables of its own type (think of a *BinaryTree* class containing instance variables *left* and *right* of type *BinaryTree*, a so-called recursive type). There is also no loop with an explicit stack. But in my Java code, you could argue, the class `GeneralContradictionStrategy` uses an instance variable referring to another strategy and this is like a recursively defined singly-linked list type. Dynamically it is configured as an 'infinite' list with itself as tail. However, we have shown in Section 4, that you could achieve the same without instance variables and just with parameters.

See (Verhoeff, 2018, §6) for more examples of 'recursion' without recursively defined functions, using self application instead and without using instance variables. See Verhoeff (2010) for a programming challenge (with interactive hints, using *Tom's JavaScript Machine*) that involves self referencing.

Acknowledgment

I would like to thank my colleagues Kees Huizing and Loek Cleophas for helping me improve this article.

References

Dijkstra, E.W. (March 1968). Go To Statement Considered Harmful. *Comm. ACM*, 11(3), 147–148.

Dijkstra, E.W. (1989). Quoted by Bob Crawford. *TUG lines*, *Journal of the Turbo User Group*, 32, Aug.–Sep.

Verhoeff, T. (2010). An enticing environment for programming. *Olympiads in Informatics*, 4, 134–141.

Verhoeff, T. (2012–2016). *From Callbacks to Design Patterns* (Version 1.7). Software Engineering & Technology, Eindhoven University of Technology, Netherlands. DOI:
`https://doi.org/10.13140/RG.2.2.28836.40320`

Verhoeff, T. (2018). A Master Class on Recursion. In: *Adventures Between Lower Bounds and Higher Altitudes*. Lecture Notes in Computer Science, Vol.11011. Springer, pp. 610–633. DOI:
`https://doi.org/10.1007/978-3-319-98355-4_35`

Verhoeff, T. (2021). Git repository with Java source code for "Look Ma, Backtracking without Recursion".
`https://gitlab.tue.nl/t-verhoeff-software/code-for-backtracking-without-recursion`
(Accessed 26 May 2021).

T. Verhoeff is Assistant Professor in Computer Science at Eindhoven University of Technology, where he works in the group Software Engineering & Technology. His research interests are support tools for verified software development and model driven engineering. He received the IOI Distinguished Service Award at IOI 2007 in Zagreb, Croatia, in particular for his role in setting up and maintaining a web archive of IOI-related material and facilities for communication in the IOI community, and in establishing, developing, chairing, and contributing to the IOI Scientific Committee from 1999 until 2007.

Olympiads in Informatics, 2021, Vol. 15, 133–141
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.11

REPORTS

The Cuban Olympiad in Informatics: A New Stage from the DMOJ Online Judge

Francisco HERNÁNDEZ GONZÁLEZ[1],
José Daniel RODRÍGUEZ MORALES[2],
Dovier Antonio RIPOLL MÉNDEZ[3]
[1]*Instituto Preuniversitario Vocacional en Ciencias Exactas de Villa Clara, Cuba*
[2]*Mantainer of DMOJ-UCLV online judge, Cuba*
[3]*General Director of ICPC Caribbean, Cuba*
e-mail: ipvce@uclv.edu.cu, josedanielr@yandex.com, dovierripoll@gmail.com

Abstract. The Cuban Olympiad in Informatics is a competition promoted by the Ministry of Education of the Republic of Cuba and among its objectives is to encourage the study of programming and algorithms in students of pre-university education. The competition has different stages ranging from the school level to the national contest. In recent years, the programming competition has been renewed with the use of an instance of the Don Mills Online Judge, an open-source online judge. This has allowed the event to be held simultaneously from all the country's provinces, which has been a challenge for students and teachers from the participating Cuban schools.

Keywords: grading systems, programming competitions, competitors, Cuban Olympiad in Informatics, informatics education, training.

1. Introduction

In Cuba, academic contests began to be developed since the 1960s, with the aim of motivating students and teachers to promote interest in studying the subjects of each discipline, encouraged by emulation with their peers in the different levels of education and regions.

The Ministry of Education of the Republic of Cuba is the main responsible for organizing, coordinating and executing knowledge and skills competitions at all educational levels.

Cuba began to develop informatics contests in education from the beginning of the 80's with the introduction of minicomputers in Vocational Schools. Then it continued with the introduction of microcomputers in the Exact Sciences high schools, and later, by stages, to all schools in the country. From the 90s on, events of this type really gained in strength in the whole country. With the annual holding of the International Olympiad in Informatics (IOI), since 1989, this activity gained in levels of organization and quality.

Currently, the academic contest format consists of different levels of competition: at the school, municipal, provincial and national levels. Based on the results in the national contest, a group of students is chosen to make up the National Preselection; in it, intensive work is being carried out to train, evaluate and finally choose the team that will represent to Cuba in the IOI and the Ibero-American Informatics and Computing Competition (CIIC by its acronym in Spanish).

The Cuban Olympiad in Informatics (OCI by its acronym in Spanish), also known as the National Computing Contest, is an event where participate the top-10 students from each province and the special municipality of Isla de la Juventud, all belonging to the pre-university education. The OCI is a competition that takes place in the first quarter of the year, simultaneously in all the country's provinces, one venue per province. It consists of two days of exam. In each one, three problems of algorithmic nature must be solved. Ten students compete in each competition venue, representing the three groups into which the pre-university is divided (tenth, eleventh and twelfth grades).

2. Informatics Contests in Cuba

In Cuba, although it has been extended to all educational levels from primary to pre-university schools, the teaching of the informatics does not include programming and algorithms, so it is necessary to teach it through complementary programs. The teaching of C/C++ programming language is the most widespread among those who show an interest in solving algorithmic problems.

The stage prior to the national contest are the provincial contests, which currently focus on solving problems of an algorithmic nature, through a programming language. The bases for its realization are defined by the group of provincial coaches and the competition is not managed by an online judge in all provinces. In the first competition stage, at the school level, it is recommended to apply math-logic tests and later train the winners in a programming language, data structures and algorithms.

Informatics contests have undergone organizational changes over the years, in search of raising the quality of work. These changes are governed by the IOI syllabus, with some adjustments between each school year. The students can participate in other complementary competitions, including the friendly-contests between Pre-University Vocational Institutes in Exact Sciences (IPVCE), as well as the ICPC university events.

The National Contest consists of two days of competition with three problems each day. Solutions are evaluated against test cases. After experimenting with different organizational variants since 2003, the contest is held in single-venues in each province of the country where a supervisor (proctor), defined by the Ministry of Education, ensures that all competition rules are met.

With the best-ranked students, the winners of the National Contest, a National Preselection is formed, which basically consists of a face-to-face training camp for intensive preparation from which the students who later represent Cuba at the IOI and the CIIC are defined. On the other side, the group of coaches who work with these students is selected according to the results of their students in national events, the academic level of the teachers and the willingness to dedicate themselves to a job that requires many hours of work per day.

Throughout this process, the link with the ICPC Competitive Programming Movement, at the university level, has been of vital importance; among the main achievements are included:

- Participation in the different stages of national university competitions.
- Participation in the Cups organized by the universities.
- Taking advantage of the experience of competitors and university coaches.
- Carrying out trainings led by university students.
- Participation of ICPC coaches in the pre-university competitions as judges.
- The use of online judges created by universities to complement the training of pre-university students.

3. Online Judges as Tools for Contestants' Trainning and Evaluation

The Online judges (Revilla *et al.*, 2008) play an important role in the preparation of the contestants, as well as in conducting programming competitions for them. They usually contain problem descriptions, an automated system for submitting and grading solutions, conducting competitions, analyzing solutions, and forums for discussing solutions from wherever the contestants are located.

The Caribbean Online Judge (COJ) is an online judge developed at the University of Informatics Sciences (UCI by its acronym in Spanish), in Havana, to contribute to the preparation of competitors/teams for the ICPC competitions. The overlapping of many of the problems in the COJ, with the topics of pre-university contests, made such a tool among the preferences for the preparation of pre-university students. Arteaga (2016) developed a module that increased the functionality of the COJ allowing partial evaluation, both in its 24-hour archive and in the execution of a new format of competitions with IOI rules.

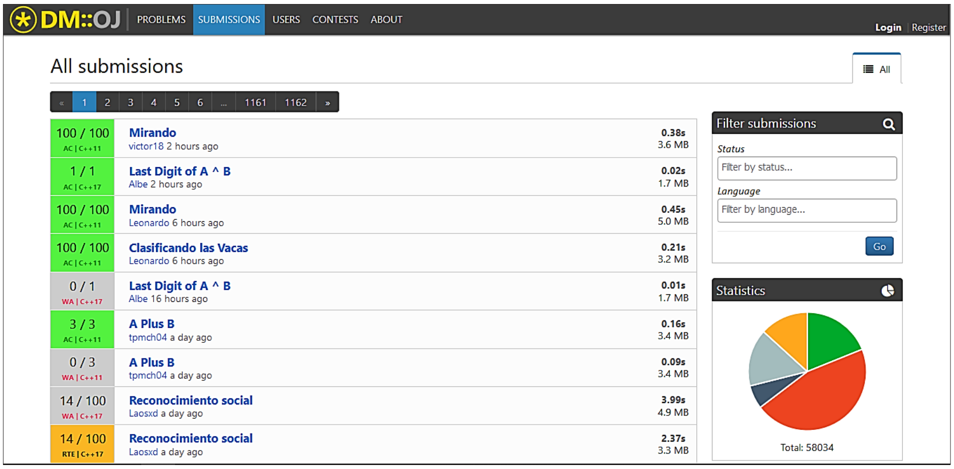
For many years, Cuban national contests were held offline and without instant feedback during the tests. The solutions grading, at the end of the competitions, was carried out manually using checker programs and a grader to verify the correct use of the running time and memory space.

The above mentioned module for IOI evaluations (Arteaga, 2016), added to the COJ, improved the grading process of the Contests and, of course, the general performance of competitions. But it was necessary to improve the efficiency in the application of the National Computing Contest, therefore an analysis of different types of judges for algorithmic competitions (Erdősné *et al.*, 2018) were done.

For this, a review of several of the existing open-source online judges were carried out, such as: HUSTOJ, Sharif-Judge, Vjudge (its license changed and it is no longer free software), Ejudge, Mooshak, BNUOJ and DMOJ. The aim was finding the most appropriate for its deployment and to be used for grading the submissions in a similar way as it is done at the IOI.

Finally, the Don Mills Online Judge[1] was chosen because it had many desirable features and also support for the pre-university competitions, such as:

- It is an active and quite mature project.
- Simple and modern interface.
- Good documentation.
- Possibility of interacting with the developers team.
- Support for a wide variety of languages.
- Good performance and stability.
- Partial scoring.
- Batch evaluation.
- Contests with time windows.
- Virtual competitions.
- Ratings system.
- Comments.



All submissions

100 / 100 AC | C++11	**Mirando** victor18 2 hours ago	**0.38s** 3.6 MB
1 / 1 AC | C++17	**Last Digit of A ^ B** Albe 2 hours ago	**0.02s** 1.7 MB
100 / 100 AC | C++11	**Mirando** Leonardo 6 hours ago	**0.45s** 5.0 MB
100 / 100 AC | C++11	**Clasificando las Vacas** Leonardo 6 hours ago	**0.21s** 3.2 MB
0 / 1 WA | C++17	**Last Digit of A ^ B** Albe 16 hours ago	**0.01s** 1.7 MB
3 / 3 AC | C++11	**A Plus B** tpmch04 a day ago	**0.16s** 3.4 MB
0 / 3 WA | C++11	**A Plus B** tpmch04 a day ago	**0.09s** 3.4 MB
14 / 100 WA | C++17	**Reconocimiento social** Laosxd a day ago	**3.99s** 4.9 MB
14 / 100 RTE | C++17	**Reconocimiento social** Laosxd a day ago	**2.37s** 3.3 MB

Fig. 1. Submissions view of the DMOJ.

[1] `https://github.com/DMOJ/online-judge`

- Editorial support.
- Language-specific time and memory constraints.
- User rating charts.
- Blog on the main page.
- Support for private competitions.

After the judge deployment it was possible to hold several competitions demonstrating its stability; later, new problems were activated for the 24-hour archive. In 2019, all the necessary regulations were made to begin to carry out the online National Computer Contest simultaneously throughout the country.

4. The DMOJ-UCLV Online Judge

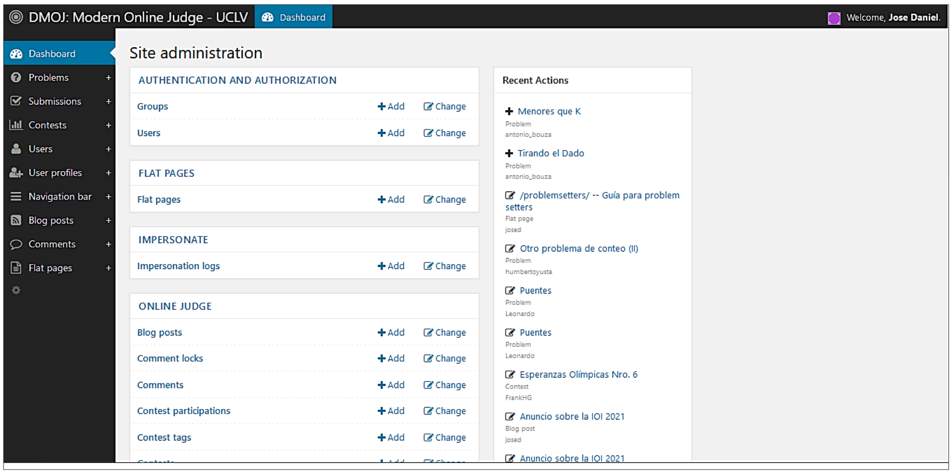
The DMOJ-UCLV[2] site has all the site's components deployed in a Virtual Private Server (VPS) in the main datacenter of the Central University "Marta Abreu" of Las Villas (UCLV by its Spanish acronym). At this time, for its operation it has 8 GB of RAM and an AMD Opteron 6136 processor with a frequency of 2.40 GHz and 8 virtual cores, but these capacities may vary depending on the demand of the site. During competitions with high user participation, such as national competitions, these capacities tend to increase and in times of low activity, such as vacations, infrastructure administrators tend to reduce resources.

The current DMOJ version is number 1, which already has ~2 years old and the dev team is working on upgrading it to the most recent version (2.1). When this process is complete, the site will be migrated to a container architecture instead of a simple virtual machine as it is now. This will make the platform easier to operate, maintain, and increase the stability.

The DMOJ has an administration interface based on WordPress which is very intuitive and is divided into several modules, among them are:

- **Problems**: for managing all the problems of privileged user since its elaboration to its publication. From there, all the memory and time constraints of a problem are defined. It allows a problem to have multiple authors, curators (a differentiation to give other administrators the possibility to refine the problem without appearing as authors) and testers (users who cannot modify the problem, but can submit and validate that test cases and restrictions are all correct).
- **Submissions**: from this module, all submissions made to the system are managed, it has several filters to reduce the list of submission to a set of interest. From here, an admin can send one or more submissions to be re-judged, as well as give a verdict manually (in problems that require it). It also includes the configuration of the languages and judges that will be available.
- **Contests**: allows the management of the competitions and their details such as date, duration, time window, problems, rating, users who may or may not participate, etc.

[2] `https://dmoj.uclv.edu.cu/`



Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+Add	Change
Users	+Add	Change

FLAT PAGES		
Flat pages	+Add	Change

IMPERSONATE		
Impersonation logs	+Add	Change

ONLINE JUDGE		
Blog posts	+Add	Change
Comment locks	+Add	Change
Comments	+Add	Change
Contest participations	+Add	Change
Contest tags	+Add	Change

Recent Actions

+ Menores que K
Problem
antonio_bouza

+ Tirando el Dado
Problem
antonio_bouza

/problemsetters/ -- Guía para problem setters
Flat page
josed

Otro problema de conteo (II)
Problem
humbertoyusta

Puentes
Problem
Leonardo

Puentes
Problem
Leonardo

Esperanzas Olímpicas Nro. 6
Contest
FrankHG

Anuncio sobre la IOI 2021
Blog post
josed

Anuncio sobre la IOI 2021

Fig. 2. Main view of the administration interface.

- **Users**: module for managing users, authentication credentials and permissions.
- **User profiles**: management of user and organization profiles. From here, the admin can view all registered users and some related metadata.
- **Blog**: management of the blog found on the main page, from here new entries are edited and the existing ones can be changed.

4.1. *System Architecture*

The site is made up of several services with different responsibilities that collaborate with each other, most of them are written in Python programming language. The main component is the web application developed with Django framework.

The other fundamental part is the judges, who are in charge of evaluating each submission sent to the online judge; they must be configured to be able to compile and execute programs written in multiple programming languages. The judges process one submission at a time, and usually multiple instances of the judges are spawned to avoid that many submissions accumulate in the queue, and also that if one of the judges fails, the system is not disrupted.

There are also other auxiliary services such as the judges' bridge and the event server. The first serves as a coordinator between the web application and the multiple instances of judges; the bridge also acts as a dispatch queue and is responsible for notifying the response backwards. The second is who allows the client to receive updates in real time in their browser, while each of the test cases is evaluated until the final verdict of the submission; this component is developed with Node.js.

The other services are typical components of any web service, such as the MySQL database engine and the NGINX web server with uWSGI.

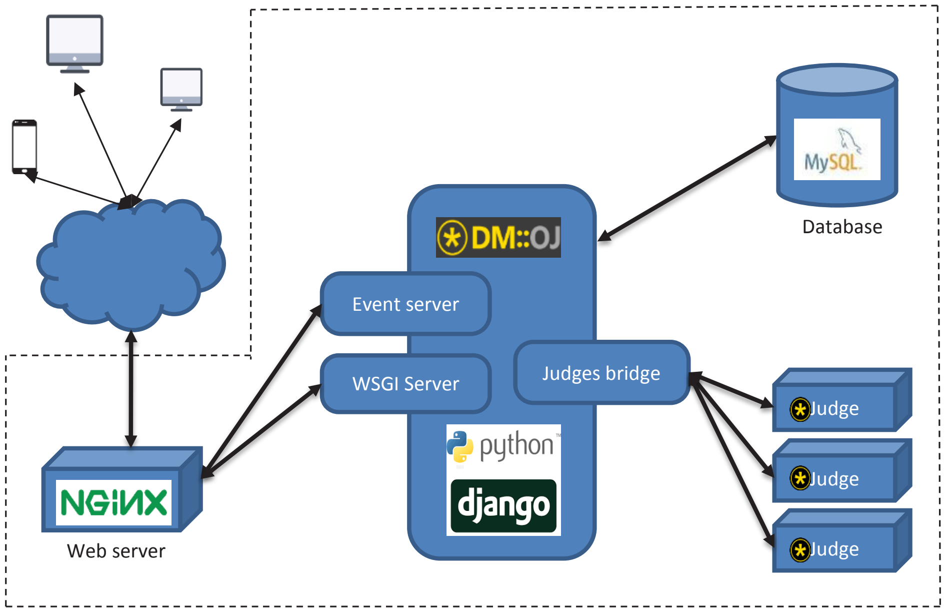


Fig. 3. Representation of the system architecture and components.

4.2. *Statistics / Facts*

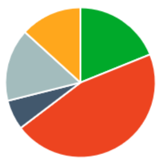
At the end of March 2021 there are 407 public problems in the DMOJ-UCLV, most of which are taken from other sources and reused in training competitions. A non-negligible part of them belongs to authentic problems published for the first time on this site, whose authors are students and professors from both universities and pre-university. Those problems have been used in official competitions such as university cups and national competitions.

There are also 633 active users and 103 competitions have been held, mainly at the pre-university level.

There are 10 programming languages available for submitting the solutions; the system maintainers are willing to add support for new requested languages. Currently supported languages are Awk, C, C++ (03, 11, 17), Assembly, Java, Lua, Pascal, Perl, Python and Scala.

A total of 58034 submissions have been made, predominantly in the C++, Java and Python languages, distributed as follows according to the verdict: .

- Wrong answer (26560)
- Accepted (10964)
- Runtime error (7689)
- Time limit exceeded (9090)
- Compilation Error (3731)



In 2019, for the first time in Cuba, a pre-university National Computer Olympiad was held online. In the year 2020, it was held on that way for the second time. The DMOJ-UCLV online judge was used on both occasions.

This constituted a milestone for students and teachers, because for the first time in OCI history all competed simultaneously from their institutions, obtaining immediate feedback on the answers sent in each problem. In addition, the online competition allowed them to know the final results of the competition within a few hours after its culmination and facilitated the work of the human judges in charge of the competition evaluation.

The experience of these two years of national online contests has been very positive. Despite having more than 150 users competing simultaneously, no instability was reported on the platform and the main difficulties occurred at the competing venues with logistic issues.

5. Conclusions

The DMOJ-UCLV has become a fundamental virtual space for high school contestants and coaches who want to train for competitions. Also, it becomes in the official host of National computer competitions and other competitive programming events in Cuban high schools.

The adoption of the online judge for the Cuban Olympiad in Informatics is a step forward to keep the contests as an attractive and motivating activity for students, while it has brought new challenges to the organizers to continue improving the quality of these events.

References

Arteaga Salgado, F.R. (2016). Módulo para juzgar soluciones con las reglas de la Olimpiada Internacional de Informática en el Juez Caribeño en Línea. *Trabajo de Diploma par a optar por el título de Ingeniero Informático.*

Erdősné Németh, Á., Zsakó, L. (2018). Grading Systems for Algorithmic Contests. *Olympiads in Informatics,*12, 159–166. DOI: 10.15388/ioi.2018.13.

Hernández González, F. (2008) "Metodología para el entrenamiento de los estudiantes de preuniversitario que participan en concursos de informática". *Tesis en opción del grado científico de doctor en Ciencias Pedagógicas Universidad Central "Marta Abreu" de Las Villas.*

Revilla M. A., Manzoor S., Liu R. (2008). Competitive Learning in Informatics: The UVa Online Judge Experience. *Olympiads in Informatics*, 2008, 2, 131–148.

F. Hernández González is an informatics teacher at the Pre-University Vocational Institute in Exact Sciences of Villa Clara. He has attended ten IOIs and has worked within the group that has coordinated the national informatics competitions. Since 1991 he has participated in the selection and preparation of Cuban students for the IOI. He received the degree of Doctor of Pedagogical Sciences at the Central University "Marta Abreu" of Las Villas for his work in the selection and development of students with programming talent.

J.D. Rodríguez Morales is a Computer Science graduate from Universidad Central de Las Villas (UCLV), Santa Clara, Villa Clara, Cuba, in 2015. He has experience as a computer science instructor at UCLV where he taught in several programming and algorithms related courses. He was involved in many ICPC activities and was coach of several teams during his time as an instructor at UCLV. He is the current maintainer of DMOJ-UCLV and collaborator of pre-university informatics events.

D.A. Ripoll Méndez has an Engineering in Informatics Sciences from the University of Informatics Sciences (UCI), Havana, Cuba, in 2008. He has experience as a Professor of Analysis and Design of Algorithms. The ICPC Executive Director for the Caribbean region, since the year 2009 to now. Co-Founder of the Caribbean Online Judge (COJ). He has been involved in the organization and realization of dissimilar activities related to the teaching of computer programming for early-age learners. He has collaborated with the inclusion of Cuba in the Bebras worldwide initiative. Since 2005, he has collaborated with the development of the pre-university Informatics Olympiads in Cuba.

Olympiads in Informatics, 2021 Vol. 15, 143–146
© 2021 *IOI, Vilnius University*
DOI: 10.15388/ioi.2021.12

Reviews of Two Programming Books

Antti LAAKSONEN
University of Helsinki, Department of Computer Science
e-mail: ahslaaks@cs.helsinki.fi

In this article I review two recent competitive programming books, published in 2020, which have not yet been presented in the *Olympiads in Informatics* journal. The books are *Algorithmic Thinking* by Daniel Zingaro and *Competitive Programming in Python* by Christoph Dürr and Jill-Jênn Vie.

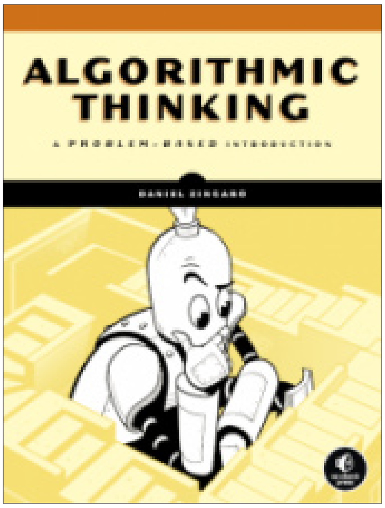
Both the books are introductory books but their approaches are different. *Algorithmic Thinking* focuses on the process of learning algorithmic problem solving and uses competitive programming problems as motivating challenges. *Competitive Programming in Python* develops skills for programming contests and job interviews, and shows how the Python language can be used in competitive programming.

Algorithmic Thinking: A Problem-Based Introduction

The book discusses data structures and algorithm design techniques, and shows how they can be used to solve problems selected from various programming contests. However, the main goal of the book is not to prepare for programming contests, but rather to teach problem solving through motivating and challenging problems. The C programming language is used throughout the book with the purpose of showing how data structures and algorithms can be implemented from scratch without using libraries.

The first chapter of the book discusses the use of hashing in algorithm design. First, the author presents a problem of classifying snowflakes and shows a quadratic algorithm for solving the problem. After sending the solution to an online judge, it turns out that the algorithm is too slow, and a better algorithm that uses hashing is developed. This is the teaching style throughout the book: first an initial algorithm is created and then it is improved step by step.

The following topics in the book are recursion and dynamic programming. Recursion is discussed in the context of tree traversal, which is first implemented using a stack and then using a recursive function. After that, the author shows how recursion can be made more efficient through memoization, which leads to dynamic programming.



Author: Daniel Zingaro
Number of pages: 408
Publisher: No Starch Press (2020)

The next two chapters focus on finding shortest paths in graphs. The book first presents a chessboard problem where an optimal sequence of moves can be found using breadth-first search. Also a more advanced problem is discussed where a special two-state breadth-first search can be used. Dijkstra's algorithm is first used to find the shortest path in a graph, and then to also calculate the number of paths with minimum length. A quadratic implementation of Dijkstra's algorithm is presented; a better implementation using a heap is included in the appendix.

Binary search is introduced using a non-standard approach which is more relevant in competitive programming: the first problem is to find an optimal solution using a function that tests whether a solution is feasible. Only after that the traditional binary search problem of locating an element in a sorted array is mentioned. Also techniques for efficiently processing static range sum queries are discussed in connection with binary search.

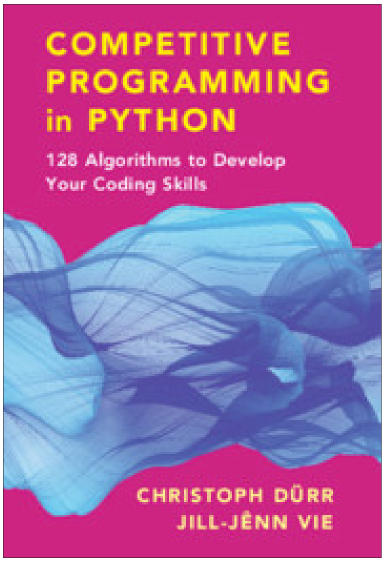
After that, the author presents two data structures that are based on a binary tree and have some similarities: a binary heap and a segment tree. While many competitive programmers use segment trees that are perfect binary trees, in this book the tree is built in a different way which does not require the size of the array to be a power of two. Finally, the last chapter of the book discusses the union-find data structure with union-by-size and path-compression optimizations.

The strength of the book is that the process of discovering and improving algorithms is described in detail and various different approaches are analyzed. Compared to traditional textbooks, there are also interesting topics that are not usually covered, including the applications of binary search, calculating the number of shortest paths using Dijkstra's algorithm, and processing range queries using segment trees.

The programming style of the author is clearer than that of most competitive programmers. However, some of the examples are difficult to understand due to low-level data structure manipulation and the old-fashioned way to declare all variables at the beginning of a function. While the purpose of using C has been to implement things from scratch, the qsort function is still used, probably because it happens to be in the C standard library.

Overall, the book is clearly written, the topics are well-chosen, and the book is a good introduction to some important competitive programming techniques. The main audience of the book are probably beginners who do not have much background in problem solving and are willing to use the C programming language.

Competitive Programming in Python: 128 Algorithms to Develop your Coding Skills


COMPETITIVE PROGRAMMING in PYTHON
128 Algorithms to Develop Your Coding Skills
CHRISTOPH DÜRR
JILL-JÊNN VIE

Authors: Christoph Dürr, Jill-Jênn Vie
Number of pages: 264
Publisher: Cambridge University Press (2020)

The purpose of the book is to teach skills that are needed for succeeding in programming contests and job interviews. The Python programming language is used in examples, which is not a typical choice in competitive programming. The reason for using Python is that it is an easy language. However, the book also mentions that it may be difficult to get Python solutions accepted, because Python is slower than languages like C++ and Java.

The book begins with a short introduction to Python and basic concepts like time complexity, data structures and algorithm design techniques. While Python provides many useful data structures in its standard library, the book contains an alternative binary heap implementation where keys can be changed. An interesting Python trick mentioned in the book is the lru_cache decorator that automatically adds memoization to a recursive function.

The next chapters present string processing algorithms, such as the KMP algorithm and Manacher's algorithm, and dynamic programming algorithms, such as calculating the edit distance of two strings and the longest increasing subsequence. An interesting example is a game where you have a stack of numbers and on each move you can either remove one number or *x* numbers from the stack, where *x* is the topmost number.

After that, the book discusses the classical maximum sum subarray problem and presents the segment tree and Fenwick tree data structures that are often used for processing range queries in competitive programming. The book also describes a data structure called interval tree that can be used to report all intervals that contain a given point.

There are four chapters devoted to graph algorithms. Most standard techniques in competitive programming are described, such as the DFS and BFS algorithms, finding shortest paths, topological sorting, and constructing an Eulerian tour. The book also discusses more advanced problems like the Chinese postman problem and the stable marriage problem, that are not often seen in programming contests.

The following chapters present techniques for processing trees, such as efficiently finding lowest common ancestors and the diameter of a tree, and more dynamic programming topics, such as finding an optimal way to construct a sum of coins. Also some geometric algorithms are discussed, including a randomized algorithm for finding two closest points. This algorithm should be easier to implement than the traditional divide and conquer algorithm.

After that, the book discusses algorithms for processing rectangles, including two important competitive programming problems: finding the maximum area of an empty rectangle in a grid and calculating the area of a union of rectangles. Finally, the book goes through some mathematical algorithms, such as calculating binomial coefficients and the sieve of Eratosthenes, and search algorithms like the dancing links algorithm that is famous for solving sudokus.

The book covers many topics, and it can be seen as a mix of standard introductory textbook topics, competitive programming topics, and more advanced theoretical topics. However, since the number of topics is large and many of them are only briefly described, it can be difficult to really understand them by reading the book. Fortunately, the book has a good list of additional literature and references that can be used to find more information.

Python is a good language for representing algorithms, but as mentioned in the book, it is often not a good choice in a programming contest – if it is available at all (for example, at the moment, Python is not available at IOI). Since most competitive programming books use C++, this book is suitable for someone who wants to use Python instead for practicing before participating in serious contests, or just for preparing for job interviews.

A. Laaksonen works as a university lecturer at the Department of Computer Science of the University of Helsinki. He is one of the organizers of the Finnish Olympiad in Informatics and has written a book on competitive programming. He is also a developer of the CSES online judge.

About Journal and Instructions to Authors

OLYMPIADS IN INFORMATICS is a peer-reviewed scholarly journal that provides an international forum for presenting research and developments in the specific scope of teaching and learning informatics through olympiads and other competitions. The journal is focused on the research and practice of professionals who are working in the field of teaching informatics to talented student. OLYMPIADS IN INFORMATICS is published annually (in the summer).

The journal consists of two sections: the main part is devoted to research papers and only original high-quality scientific papers are accepted; the second section is for countries reports on national olympiads or contests, book reviews, comments on tasks solutions and other initiatives in connection with teaching informatics in schools.

The journal is closely connected to the scientific conference annually organized during the International Olympiad in Informatics (IOI).

Abstracting/Indexing

OLYMPIADS IN INFORMATICS is abstracted/indexed by:

- Cabell Publishing
- Central and Eastern European Online Library (CEEOL)
- EBSCO
- Educational Research Abstracts (ERA)
- ERIC
- INSPEC
- SCOPUS – Elsevier Bibliographic Databases

Submission of Manuscripts

All research papers submitted for publication in this journal must contain original unpublished work and must not have been submitted for publication elsewhere. Any manuscript which does not conform to the requirements will be returned.

The journal language is English. No formal limit is placed on the length of a paper, but the editors may recommend the shortening of a long paper.

Each paper submitted for the journal should be prepared according to the following structure:

- concise and informative title
- full names and affiliations of all authors, including e-mail addresses
- informative abstract of 70–150 words

- list of relevant keywords
- full text of the paper
- list of references
- biographic information about the author(s) including photography

All illustrations should be numbered consecutively and supplied with captions. They must fit on a 124 × 194 mm sheet of paper, including the title.

The references cited in the text should be indicated in brackets:

- for one author – (Johnson, 1999)
- for two authors – (Johnson and Peterson, 2002)
- for three or more authors – (Johnson *et al.*, 2002)
- the page number can be indicated as (Hubwieser, 2001, p. 25)

The list of references should be presented at the end of the paper in alphabetic order. Papers by the same author(s) in the same year should be distinguished by the letters a, b, etc. Only Latin characters should be used in references.

Please adhere closely to the following format in the list of references:

For books:

Hubwieser, P. (2001). *Didaktik der Informatik*. Springer-Verlag, Berlin.

Schwartz, J.E., Beichner, R.J. (1999). *Essentials of Educational Technology*. Allyn and Bacon, Boston.

For contribution to collective works:

Batissta, M.T., Clements, D.H. (2000). Mathematics curriculum development as a scientific endeavor. In: Kelly, A.E., Lesh, R.A. (Eds.), *Handbook of Research Design in Mathematics and Science Education*. Lawrence Erlbaum Associates Pub., London, 737–760.

Plomp, T., Reinen, I.J. (1996). Computer literacy. In: Plomp, T., Ely, A.D. (Eds.), *International Encyclopedia for Educational Technology*. Pergamon Press, London, 626–630.

For journal papers:

McCormick, R. (1992). Curriculum development and new information technology. *Journal of Information Technology for Teacher Education*, 1(1), 23–49.
`http://rice.edn.deakin.edu.au/archives/JITTE/j113.htm`

Burton, B.A. (2010). Encouraging algorithmic thinking without a computer. *Olympiads in Informatics*, 4, 3–14.

For documents on Internet:

International Olympiads in Informatics (2008).
`http://www.IOInformatics.org/`

Hassinen, P., Elomaa, J., Ronkko, J., Halme, J., Hodju, P. (1999). *Neural Networks Tool – Nenet (Version 1.1)*.
`http://koti.mbnet.fi/~phodju/nenet/Nenet/General.html`

Authors must submit electronic versions of manuscripts in PDF to the editors. The manuscripts should conform all the requirements above.

If a paper is accepted for publication, the authors will be asked for a computerprocessed text of the final version of the paper, supplemented with illustrations and tables, prepared as a Microsoft Word or LaTeX document. The illustrations are to be presented in TIF, WMF, BMP, PCX or PNG formats (the resolution of point graphics pictures is 300 dots per inch).

Contacts for communication

Valentina Dagienė
Vilnius University
Akademijos str. 4, LT-08663 Vilnius, Lithuania
Phone: +370 5 2109 732
Fax: +370 52 729 209
E-mail: valentina.dagiene@mif.vu.lt

Internet Address

All the information about the journal can be found at:

`https://ioinformatics.org/page/ioi-journal`

Publisher office: Vilnius University
Akademijos str. 4, LT-08663 Vilnius, Lithuania
June, 2021

Olympiads in Informatics

Volume 15, 2021

Volume 15, 2021

ISSN 1822-7732