Olympiads in Informatics, 2025, Vol. 19, 87–100

DOI: 10.15388/ioi.2025.06

Pisek – a Caching Task Preparation System

Martin MAREŠ, Daniel SKÝPALA
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, skipy@kam.mff.cuni.cz

Abstract. We introduce a new tool for developing competition tasks. It helps with creating test data and checking that the tests award the expected scores to a set of reference solutions. It supports batch, interactive, and open-data tasks in a variety of programming languages. Test results are cached, which significantly accelerates task development. Automated checks are utilized to detect common errors, including fuzzing of output checkers. The tool interfaces to CMS for configuring tasks, testing them, and semi-automatically establishing time limits.

Keywords: task preparation tool, automated testing.

1. Introduction

Preparing a task for a programming competition is an elaborate process, which includes developing the task statement, creating test data, and checking that the test data award the expected scores to a set of reference solutions. Experience shows that this process is prone to errors, especially when last-minute changes are introduced in a hurry.

Contest organizers therefore strive to make the task preparation process rigorous. One such process was documented by Diks *et al.* (2008) and its principles are still followed by major contests.

An immediate consequence is the development of task preparation systems that try to automate as much of the process as possible. They take a formal description of the task, its tests, and reference solutions. Then they go through all steps of the process and check for errors. Some steps still require human intervention, for example setting of time limits. But even there, the task preparation system can provide guidance.

There already exist multiple task preparation systems, most notably Polygon[1] (popular at CodeForces), TPS[2] (developed for IOI 2017), sinol-make[3] (originated in the Polish OI), and Taskmaker[4] (originated in the Italian OI).

[1] `https://polygon.codeforces.com/`
[2] `https://github.com/ioi/tps`
[3] `https://github.com/sio2project/sinol-make`
[4] `https://github.com/olimpiadi-informatica/task-maker-rust`

In this paper, we present Pisek[5] – a system we have developed over the past few years. It is powerful and fast, while being very simple with minimal dependencies. In particular, it can be easily used by task authors on their own machines. The current version of Pisek is available at `https://github.com/piskoviste/pisek/`.

We aim for supporting a much wider range of contest types and task formats – in particular, both IOI-type contests where solutions are submitted as source code, and open-data contests where the contestants download test inputs and submit the corresponding outputs. We also support a wide variety of programming languages.

Pisek is based on its own task format, which tries to make common things straightforward and less common things possible. Tasks developed in this format can be later exported to an actual contest system.

Pisek has a simple command-line interface, which can be used manually or invoked as a part of a continuous integration system. Pisek employs a lot of caching behind the scenes to make development cycles short while ensuring correctness.

Inside, Pisek is implemented in as a collection of Python modules that can also serve as building blocks of other tools for handling tasks, or even of contest systems.

This paper presents the features of Pisek and the foundations on which it is built. Section 2 introduces the task format and the components of the task development process. Section 3 describes deeper layers, in particular handling of programming languages and the caching layer. Section 4 discusses integration with contest systems like the CMS.[6]

1.1. *History of Pisek*

The first version of Pisek was developed in 2019 by Jiří Beneš, Richard Hladík, Michal Töpfer, and Václav Volhejn for a Czech open-data contest called Kasiopea,[7] drawing inspiration from the KSP open-data system[8] developed by Martin Mareš. Then it was extended to handle IOI-type tasks for the Czech IOI team selection camp.

Between 2023 and 2025, Pisek was rewritten by Jiří Kalvoda, Daniel Skýpala, and Benjamin Swart, based on experience with the initial version and further ideas by Martin Mareš. This version is described in this paper. It is also used to develop tasks for the Czech national programming olympiad and CEOI 2024.

2. Tasks and their Parts

First of all, we introduce the underlying concepts of tasks and their testing. Then we explain how these concepts are expressed in Pisek.

[5] "písek" is a Czech word for sand, alluding to a playground for children.
[6] `https://github.com/cms-dev/cms`
[7] `https://kasiopea.matfyz.cz/`
[8] `https://ksp.mff.cuni.cz/`

2.1. *Anatomy of a Task*

Pisek supports two types of tasks: *batch tasks* (the solution is a single program that reads an input and then produces the corresponding output) and *interactive tasks* (a program that interacts with the contest system in multiple steps; e.g., a two-player game). By default, all communication is performed via the standard input and output, but the task can define a library that wraps such communication in an arbitrary API provided to the solution.

The goal of the task is specified in a *task statement* given to contestants. Statements are not handled by Pisek.

Solutions are graded using a set of *tests*, each having one or more *testcases*. Each test is worth a certain amount of points, which are awarded for solving all testcases in the test. For IOI-style tasks, tests correspond to subtasks. Sample input and output (given openly to the contestant) is also considered a separate test.

In a batch task, a testcase specifies an input to the solution and the correct output. The input can be a static file, but it is usually created using a *generator*. The correct output can be static, but it is often computed from the input using a correct *primary solution*. A *checker* then decides if the solution's output matches the correct output. It can be a diff-like program, or if there are multiple correct outputs, the task can provide a *judge program* for checking correctness. The judge may also award partial score (e.g., in optimization tasks), the total score per test is then computed as the minimum over all testcases.

In an interactive task, there is always a judge program, which interacts with the solution over a pair of pipes. There is also an input file, but it is consumed by the judge. Again, the judge may award partial score.

A task also comes with several *reference solutions* with expected scores. One of the solution is declared *primary*. A primary solution is expected to solve all testcases correctly and efficiently.

In addition to solutions, a task can define a *validator*. It is a separate program that meticulously verifies that the input files conform to the format set in the task statement. In some cases, validation is integrated in the primary solution instead.

Generators, judges, and validators can have access to a *dataset* – a collection of data files that are either contained in the task package or generated by a separate program.

2.2. *Task Package*

Pisek represents everything related to a single task as a *task package*. The package is stored as a single directory in the file system (possibly with subdirectories). The contents are typically maintained in a Git repository, but Pisek is oblivious to versioning.

Behavior of the task is controlled by a *configuration file* with a simple INI-like syntax (essentially a collection of key-value pairs divided to sections) – see Fig. 1 for an

```
[task]
version=v3           # Config version, v3 is the newest one.
use=ceoi2024         # Use contest-specific default settings.

[tests]
in_gen=gen           # Program used to generate inputs.
validator=validate   # Program used to validate inputs.
out_check=tokens     # Compare outputs using token-based checker.

[solution_model]
primary=yes          # This sol. used for generating correct outputs.
tests=1111           # It should succeed on all tests.

[solution_greedy]
tests=W1WW           # Succeed only on test01, otherwise wrong answer.

[solution_slow]
tests=111T           # Timeout on test03 and succeed otherwise.

[solution_segfault]
tests=X!!!           # Any result on samples, runtime error on the rest.

[test01]
points=2             # This test is for two points.
in_globs=01*.in sample1.in   # Inputs assigned to this test.

[test02]
points=3
in_globs=02*.in sample2.in

[test03]
points=5
in_globs=03*.in
predecessors=1 2     # This test is a superset of both test01 and test02.

[run_solution]
time_limit=1         # Time limit for solutions.

[cms]                # Data for CMS importer.
name=sandcastle
title=Building sandcastles
time_limit=1
```

Fig. 1. An example configuration file.

example. The configuration can refer to a parent configuration file that supplies defaults for non-specified items. Typically, the parent configuration is specific to a contest. The ultimate parent is the set of defaults provided by Pisek itself.

The task package also contains a collection of static input and output files and source code of all programs related to the task (generators, validators, judges, reference solutions etc.).

Finally, there may be extra files not handled by Pisek. This typically includes the task statement.

2.3. *Generators*

In addition to static testcases, task authors can implement a *generator* that produces further testcases in a mechanic way. Pisek supports multiple generator interfaces, but all of them follow the same logic:

- **The generator is deterministic** – the generated input file depends only on the generator itself, its runtime arguments, and possibly on the dataset. If the generator uses pseudo-random numbers, it should fix their *random seed* to one provided in the runtime arguments. This is crucial for reproducibility of testing and Pisek's caching.
- **The generator respects the seed** – for different seeds, the generator should generate different input files. This is especially useful in open-data contests where each attempt to solve the task produces new input data based on a fresh seed, which expires after some time. It is also possible to declare that a particular test does not have a seed.

The mapping of tests to testcases depends on the particular generator interface. In the trivial case, each test has a single testcase named after the test.

With the more advanced interfaces, the generator can be asked to produce a list of testcases it can generate. Each testcase has a file name (e.g., `easy01.in`) and optional attributes: if it is seeded and how many instances of the testcase (with different seeds) should be generated.

The configuration file can then specify a list of filename globs for each test, e.g., `in_globs=01*.in easy*.in`. All testcases (static and generated) matching any of the globs are included in the test.

Moreover, a test can also define one or more *predecessor tests*, whose testcases are automatically included. For example, the contest-specific configuration can specify that the predecessor should be the previous test. Transitively, this makes each test to include its own testcases and testcases of all previous tests.

2.4. *Checkers*

A batch task needs a *checker* to decide if the solution's output is correct. Pisek provides a variety of built-in checkers that compare the solution's output with the correct output at different levels of strictness:

- **Diff** – runs the `diff` utility provided by the operating system, set to ignore differences in whitespace. This is a traditional method, but it suffers from quadratic time complexity in the worst case.
- **Tokens** – compares the two outputs as sequences of whitespace-separated tokens. By default, newline characters are considered separate tokens, but the task can choose to make them equivalent to other whitespace. Additionally, the checker can be configured as case-insensitive and/or to compare numeric tokens with a given precision. This checker is the recommended choice if the correct output is unique up to formatting.
- **Shuffle** – a token-based checker that accepts all permutations of tokens within a line, or all permutations of lines within a file, or both. It is useful if the correct output is unique up to order.

If there are multiple correct outputs (e.g., multiple shortest paths in a graph), the task provides a custom *judge*. Pisek supports several interfaces to judges, including the one used in CMS.

Depending on the interface, the judge can be given the test number, the seed used to generate the input, the input, the correct output (as produced by the primary solution), and the solution's output. The input and the correct output are optional – some judges do not need them, as they can compute everything from the seed. This is useful if Pisek is used within an open-data contest system, which can skip generating the unneeded files and save time.

The main part of the judge's output is the *verdict* (accept or reject, possibly with a message for the contestant). Optionally, the judge can award points (absolute or relative to the number of points per test).

Interactive tasks always require a custom judge, which talks to the solution over a pair of pipes. Pisek currently supports only the manager interface of communication tasks in CMS. The judge gets the input and produces a verdict as with the batch judges.

In the future, we plan to design a more flexible interactive judge interface, because the CMS interface suffers from multiple problems. In particular, use of named pipes leads to deadlocks if they are opened in an unexpected order. Furthermore, it is not possible to report wrong answers differently from protocol errors, which leads to confusing results if the protocol error is caused by a pipe being closed automatically after the solution crashes. This is in need of more research and hopefully also cooperation among maintainers of contest systems.

2.5. *Solutions*

The task specifies a *primary solution* and an arbitrary number of *secondary solutions*. The primary solution should be correct and efficient; it is used to produce the correct output if the checker needs it. The set of secondary solutions usually includes other correct solutions (to ensure that the primary solution is correct) and also incorrect solutions with a wide range of mistakes (to ensure that the scoring strategy works as expected).

Solutions communicate over their standard input and output, although this can be wrapped in a library (see below). Solutions typically have their running time and memory limited.

For each solution, the task configuration specifies the expected outcome. It can be the expected number of points or the expected outcome for each test (e.g., test 1 passes, test 2 produces a wrong answer, test 3 times out). The expected outcomes are preferred, but expected points can be more useful in optimization tasks.

2.6. *Verification*

There are many possible mistakes in competition tasks, but they frequently follow one of a few typical patterns. Pisek provides a battery of checks for such common errors. All of them are optional, defaults are usually provided by the per-contest configuration.

- **Size of inputs and outputs** – sizes are compared with a configured maximum. This can catch a run-away generator. In open-data contests, the limits are usually more strict, because the contestants must be able to download the input, run their program, and upload its output within a short time window.
- **Coverage of tests by solutions** – for each test, there should be a reference solution that succeeds on this test and all its predecessors, but fails on all other tests. This is useful if subtasks of the task are linearly ordered (each is a strict superset of the preceding one) or if their dependencies form a rooted tree.
- **Unused inputs** – every input (static or produced by the generator) should be included in at least one test.
- **Last test uses all inputs** – if the subtasks are linearly ordered, the last test should include all inputs.
- **Generator depends on seed** – the generator produces different input files for the same testcase with different seeds. This can produce false positives in tasks with short inputs, but our experience shows that it is rare in practice.
- **Fuzzing** – if the task has a custom judge, this check tries to run it on many randomly mutated copies of the sample outputs. This often crashes judges with sloppy parsing of the solution's output.

Additionally, a validator supplied with the task is ran on each testcase. Its goal is to check conformance of the input to the task statement. The validator is also given the test where the testcase belongs, so it can verify properties required by specific subtasks.

2.7. *Preprocessors*

Input and output of most tasks is a simple ASCII text. But the simplicity is often deceiving: text files can contain trailing spaces at the end of a line, multiple spaces in a row, or tabulators instead of spaces. Lines can be terminated by different newline characters,

the final newline can be missing, or perhaps there are a few extra empty lines at the very end. Windows programs tend to add the UTF-8 byte-order marker at the beginning of text files, even if the text contains only ASCII characters. Sometimes, they also encode the ASCII text in UTF-16.

Some of these problems are unknown in the C++-centric world of major competitions. But once a competition enables more exotic programming languages, or if the tasks are open-data, all of them become everyday issues.

Handling all these anomalies in checkers and judges is a tedious task prone to errors. Pisek avoids problems with irregular whitespace by using token-based checkers (and we provide a tokenization library to custom judges). To handle the other problems, Pisek runs all text files through a preprocessor that normalizes character encoding and newline characters (including proper termination of the last line).

Preprocessing takes place in three situations:

- **All inputs (both static and generated)** – the inputs are normalized first. If an input contains non-ASCII or non-printable characters, normalization fails and so does testing of the task. If normalization changes the input, depending on the configuration either the normalized input is used instead, or an error is raised.
- **Outputs produced by solutions** – they are normalized before they are checked for correctness. A warning can be also produced if the output was non-normalized. Failed normalization causes the testcase to fail.
- **Outputs produced by contestants** – if Pisek is used as a part of an open-data contest system, outputs uploaded by contestants are also normalized.

Tasks with non-ASCII input/output can set their input/output format to binary and check correctness using a judge. New formats can be added easily. An obvious candidate is Unicode text in UTF-8 or UTF-16, but that would bring a completely new set of normalization issues (see Whistler (2024)).

Preprocessing does not take place for interactive tasks. Their judges must cope with non-normalized text.

3. Building and Running Programs

Development of a task involves running different programs: generators, validators, judges, and reference solutions. They are written in varying programming languages. First, it is good practice to test solutions in all languages available to the contestants, so that time limits can be calibrated accordingly. Second, task authors often prefer to use higher-level languages (e.g., Python) for generators and validators, which need not run quickly.

Let us consider typical use cases first:

- **Simple C++** – The task package contains one source file. We need to run a compiler, which produces an executable file. In some cases, the task author wants to add custom compiler options or to link a well-known library. Most traditional compiled languages also fall into this category.

- **Simple Python** – The task package contains one source file. We can run it directly. This also applies to languages like Perl, Ruby, Raku, and JavaScript.
- **Simple Java** – The task package contains one source file. We need to run a compiler, which produces byte code. To run it, we need to invoke the Java virtual machine. Alternatively, we can set up `binfmt_misc` on Linux to make the kernel recognize the byte code signature and run the JVM automatically. We prefer to avoid this approach, because it needs root privileges and we cannot adjust JVM options per task. A similar case is C#.
- **Multi-file C++ or Java** – Like Simple C++, but we have multiple source files which have to be compiled and linked together to produce a single binary.
- **Multi-file Python** – We have multiple source files, but no compiler. All files have to be present when running the program. An alternative is to use the little-known `zipapp` module from Python's standard library that can pack all files to a single ZIP archive which is then runnable by the Python interpreter. We still need a generic solution for other Python-like languages.
- **Rust with Cargo** – Rust programs are usually built using Cargo from a directory with all source files and a configuration of Cargo. A similar case is Go with its module system.
- **Make** – In rare cases, there is a program with a complex building process. It can be a multi-language program, or perhaps a program whose source code is generated by another program. As we do not want to implement yet another universal build system, we prefer to defer to an existing build system in such cases. For sake of tradition, let us consider Make. The source code is then a directory with a `Makefile`.
- **Task-specific stubs and libraries** – At some contests (e.g., recent IOIs), solutions are expected to implement an API instead of communicating using files. The solution is then linked with a *stub*: a piece of code specific for a combination of a task and a language that serves as the interface between the contest system and the API. Usually, the stub reads the input from the standard input, calls the solution's API, and writes the result to the standard output. Similarly, an interactive task can provide an API called by the solution to interact with the judge.
- **Multi-purpose binaries** – Sometimes, we want to share code among generators, judges, and validators. A single source file can participate in compilation of multiple binaries. Or we can produce a single binary which can play multiple roles, depending on the command-line arguments passed.

Overall, we want to handle the simple cases (e.g., a single C++/Python source file) with as little configuration as possible, while still allowing the complex cases.

This is accomplished by two parts of configuration: *build sections* that describe how programs are compiled from their sources, and *run sections* that specify how the programs should be run. All settings in these sections have defaults such that in the typical case, you can omit the sections completely and just specify the name of the program.

3.1. *Building Programs*

A build takes the source and produces an executable program. The *source* is either a single file or a sub-directory. The *executable program* is either a single file executable by the OS or a sub-directory containing an executable file called `run` that can refer to the rest of the sub-directory (relative to the path it was ran from).

The build is governed by a *build section* in the task configuration. The section is named after the combination of a program name and its purpose, e.g., `[build_solution:good1]`. It specifies the name of the source and a *build strategy* to be used. Available strategies include:

- **A simple C++ program** – compiles a single source file to a single executable file.
- **A simple Python program** – just copies the source file and marks it as executable.
- **A simple Java program** – compiles a single source file to a byte code file, produces a directory, were `run` is a shell script that runs the JVM on the byte code.
- **A multi-file Python program** – takes a directory and a given entry point, produces a directory with `run` symlinked to the file with the entry point.
- **Cargo** – takes a directory and runs Cargo in it to produce a single file.
- **Make** – takes a directory, runs `make` in it; the `Makefile` is supposed to produce output in a sub-directory called `target`, which contains either a single executable file or a collection of files with an executable `run`.

If neither the source nor the strategy is given, Pisek chooses automatically. Most strategies have an auto-detection rule. For example, if we are building the solution `good1` and the task package contains a file `good1.cpp`, the C++ strategy is willing to build it. If multiple strategies match, an error is raised and the user must make an explicit choice. So in the simple cases, the whole build section can be omitted.

Additionally, the build section can set strategy-specific options like compiler options, further files to be made available to the compiler (e.g., header files) and additional source files to be compiled together with the main source file (e.g., task stubs). This is useful in conjunction with inheritance of build section: `[build_solution:good1]` inherits from `[build_solution]` (e.g., task-specific libraries) and `[build]` (e.g., compiler flags provided by contest-specific configuration).

3.2. *Running Programs*

Whenever task configuration specifies a program to be run (e.g., a solution), it actually refers to a *run section* named after the program and its purpose. For example, `[run_solution:good1]`. The run section refers to a build section that produces the program and it specifies the command-line arguments to be passed and resource limits to be applied (e.g., a time and memory limit).

Again, there are defaults that allow omitting the whole section: we build `[build_solution:good1]` and run the program with no arguments. There is an inheritance

hierarchy of `[run_solution]` and `[run]` that typically provides resource limits. For solutions in particular, we also inherit from `[run_primary_solution]` and `[run_secondary_solution]`, which is often used to run secondary solutions with a less strict time limit.

3.3. *Sandboxing*

Programs should be run within a sandbox that imposes resource limits and checks that the programs access only the expected files (this is important to ensure consistent caching).

Pisek currently uses `minibox`, a simple pseudo-sandbox which limits memory using the kernel's `ulimit` for virtual memory and which kills the program when the time limit is exceeded. It is not a proper sandbox as it is easy to escape from it. But it is actually sufficient in most cases as the programs in the task package can be trusted not to be malicious. (However, beware when using somebody else's task packages.)

The advantage of this approach is simplicity and no need for root privileges. Disadvantages include problems with limiting memory in C# and Go (both runtimes allocate enormous amounts virtual address space without actually using it) and the impossibility of controlling programs with multiple processes or threads.

In the future, we plan to switch to Isolate (Blackham and Mareš, 2012)) and/or `systemd-run` (weaker, but available in most Linux distributions by default).

3.4. *Caching*

Testing a task in Pisek can be a time-consuming process. We need to generate all input files, validate them, run all verification checks, run all solutions, and check their output. All this can easily take at least minutes for an IOI-level task. On the other hand, it is good practice to re-test the task after every change, especially in the later stages of contest preparation.

We observe that minor changes in the task often affect only a small subset of Pisek's operations. We can therefore save significant time by caching results of operations and re-computing them only if the relevant parts of the task change.

This is similar to what build systems like `make` do, but they need the user to declare explicit dependencies, which is prone to errors. We prefer a systematic and automated approach that is as close to obviously correct as possible.

Testing of tasks is divided to small pieces called *jobs*. Each job can depend on results of other jobs, called its *prerequisites*. There is a universal mechanism for caching job results. Each *cache entry* contains the following information:

- *Name* – a human-readable description of the job (e.g., "Run solution *name* on input *name*").
- *Result* – the output of the job (e.g., if running the solution succeeded).

- *Signature* – a cryptographic hash of all data on which the job depends. This includes:
  - `__init__` *arguments* – each job is internally a class, whose initialization parameters specify what the job should do (with more details than what is specified in the job's name).
  - *Results of prerequisites.*
  - *Testing context* – values of command-line arguments and all settings in task configuration which have been accessed when the job was run.
  - *Contents of files* – for each file read or created by the job, we record the hash of its contents, which is then added to the collective signature.
  - *Evaluation of globs* – if the job uses filename globbing (e.g., to select testcases for a given test), we need to check that the glob still produces the same set of files. Otherwise, dependencies on file contents would not catch a newly matched file.
- *Signature recipe* – a list of all inputs from which the signature was computed.

The cache can contain multiple entries with the same job name, but different signatures. (This is why the signature covers contents of files produced by the job: Different versions of the job may have the same output file with different contents.)

When Pisek wants to run a job, it looks up all entries with the right name in the cache. For each such entry, it computes the signature according to the entry's recipe. If it matches the entry's signature, the job is considered unchanged and the cached result is re-used. If the job needs recomputing, a new entry is created with the same name and a new signature. If there are too many entries with the same name, we trim the oldest ones.

The jobs are fine-grained, which enables Pisek to recompute only the absolute minimum when the task changes. For example, when we change the judge, we do not re-run the solutions and we only re-judge their outputs. When a new testcase is added, solutions are run only on that testcase etc.

This systematic approach has proven itself efficient and reliable. Over the years we used Pisek, there were very few errors, usually caused by colliding job names or file names. The cache is automatically invalidated when Pisek is upgraded to avoid compatibility errors. (However, this does not apply when using the development version of Pisek from its Git repository as the version number changes only for official releases.)

4. Integration with Contest Systems

When the task is tested in Pisek, we need to export it to the actual contest system. The export should be automated to the greatest extent possible to avoid human errors.

The environment in which the solutions run within the contest system is obviously different from the environment used by task authors. So we need to verify that the behavior of tasks in the contest system matches the expectations.

4.1. *Integration with CMS*

We have implemented an export to CMS, which can set up the task, create a dataset with the test data, set time and memory limits, submit all reference solutions, download test results, and compare them to the expected results.

We also have a semi-automatic tool for choosing the time limit. This requires detailed specification of the expected behavior of the reference solutions on tests. In particular, we need to separate timeouts from the other failure modes. Then we can compute the time interval between the slowest solution that should not time out and the fastest one that should time out. The time limit is then chosen manually from the computed interval. If the interval is empty, we must improve the test data.

Since Pisek supports a much wider variety of tasks, there are some restrictions. The task must use judge interfaces compatible with CMS and it cannot rely on the text preprocessor.

Currently, the CMS lacks a public API for creating tasks and submitting solutions. Our CMS interface therefore relies on CMS internals and calls CMS libraries in ways that can break in the future. We will try to keep up with changes in CMS, but the proper solution is to make CMS offer a well-defined API.

4.2. *KSP Open-data System*

With the KSP open-data contest system, we plan a completely different approach. We are going to re-implement the back-end of the contest system on the top of Pisek. Most of the necessary functionality is already available in Pisek as separate modules: most importantly generating the input data for a given seed and testing if the output is correct.

The only significant difference is that we have to separate the actions performed online during the contest (generating inputs and checking outputs) from those that take place when setting up the task (compiling programs, preparing datasets).

5. Conclusions

Pisek has proven itself useful when developing tasks for multiple contests including CEOI 2024.

Still, there remain several areas which call for further research and development. Most importantly, we would like to extend compatibility between Pisek and CMS: support the full range of Pisek's built-in checkers, judge interfaces, and possibly also the text preprocessor. One possibility is to improve CMS itself, another is auto-generating manager code for CMS.

The task format could be brought closer to the Kattis problem package specification.[9] Both formats would benefit from cross-pollination and automated conversion of tasks between them.

A task statement (and its translations) could be added to the task package format, which would enable automatic inclusion of sample inputs and outputs. Formalizing descriptions of subtasks (at least partially) could enable sharing a single definition of limits among the task statements, the validator, and possibly also the generator. The task-maker already supports similar features.

Further automated checks for common errors should be included, especially a more powerful fuzzer.

Judges and validators of different tasks contain a lot of common code, often implemented with insufficient handling of malformed inputs. We suggest that this common code should be generalized and made available as a library. The library should be independent of the task preparation system used.

References

Blackham, B., Mareš, M. (2012). A New Contest Sandbox. *Olympiads in Informatics*, 6, 100–109.
Diks, K. *et al.* (2008). A Proposal for a Task Preparation Process. *Olympiads in Informatics*, 2, 64–74.
Whistler, K. (ed.) (2024). Unicode Standard Annex #15: Unicode Normalization Forms. Available online at: https://unicode.org/reports/tr15/

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.

D. Skýpala is a Bachelor student at Faculty of Mathematics and Physics of the Charles University in Prague, IOI 2022 bronze medalist, organizer of several Czech programming contests and a member of CEOI 2024 Scientific Committee.

[9] https://github.com/Kattis/problem-package-format