Scrool Problem Creation Guide

Per Austrin

Contents

1 Introduction

This document documents how to construct and test a problem package.

Hopefully you also have access to an example problem package called `different`, and an empty problem skeleton `problem_skeleton`. On most occasions it is probably easier to just start from the problem skeleton, and refer to the README files contained therein (which contain large portions of this guide). In case of doubt, the `different` package should be useful. If anything remains unclear, this document is here to help (hopefully).

1.1 Overview

All files related to the problem resides in a directory `<shortname>`, where `<shortname>` is some identifying short version of the problem name (e.g. `different` if the problem is called "A Different Problem". The shortname must be lower case and use only the characters 'a'-'z' and '0'-'9'.

The directory structure is as described below. Items enclosed by brackets ([...]) are optional, everything else is required.

```
<shortname>/
    problem.yaml - problem configuration file
    problem_statement/
        problem*.tex - problem statements
        (not including sample data; it is included automatically)
        - auxiliary problem statement files, e.g., images
    data/
        [sample/] (optional but gives warning if missing)
            *.in - sample input files
            *.ans - sample answer files
        secret/
            *.in - input files
            *.ans - answer files
            [*.txt] - data file description
    submissions/
        accepted/
            - correct solutions
        [wrong_answer/]
            - incorrect solutions
        [time_limit_exceeded/]
            - too slow solutions
        [run_time_error/]
            - crashing solutions
    input_format_validators/
        - input format validators
    [output_validators/]
        - output validators, if necessary
```

The remainder of this document describes the precise content of each of these directories.

1.2 Tools

A few tools are available to test a problem package. These tools are available from ???. The tools are still in beta, and are likely to have bugs. If you find such a bug, report it to `austrin@scrool.se` and we'll try to fix it.

The most important tool is verifyproblem. It runs a complete check on a problem package, and informs you of any errors or warnings.

Two other useful tools are problem2pdf and problem2html. These convert the problem statements to the respective formats. When you run problem2pdf you get the output from pdflatex, which, though it is hard to parse at times, may hopefully allow you to debug errors in your LaTeX source. The problem2html can be used to check that the problem can be converted to html successfully.

1.3 How to Supply Programs

This section describes how programs (both sample solutions and validators) are provided in the problem package.

There are essentially three ways of providing programs: as single files, as a directory, and as a directory with build and run scripts.

1.3.1 Single-file programs

Single-file programs that do not require any specific compiler options can be provided directly in the respective directories (e.g., `submissions/accepted/different_pa.cc`).

1.3.2 Multi-file programs

If the program consists of more than one file, all the files of the program needs to be placed in a separate subdirectory (e.g., a directory `submissions/accepted/different_pa` containing two files `different_pa.java` and `io.java`).

Such a program will be compiled as if all the files in the directory were submitted (so for instance if there are three files `defs.h`, `sol.cc`, `data.cc` it will be compiled as `g++ sol.cc data.cc`).

1.3.3 Programs with build and run scripts

If there is something non-standard about the way the program should be compiled or run, the program can be provided in a directory containing two scripts `build` and `run`. When compiling the program the `build` script will be used, and when running the program the command `run` will be used. Note that `run` does not have to be a provided script, it could be created by the `build` script.

```
source: ICPC Mid-Atlantic Regional Contest
author: John von Judge
rights_owner: ICPC
```

Fig. 1: A typical problem.yaml

2 problem.yaml – metadata and configuration

the file `problem.yaml` contains configuration and metadata for the problem. It is a YAML file with the mappings described below. Example problem.yaml files are shown in Figures 1 and 2.

Key	Description
`source`	Where the problem was first used (optional)
`author`	Author of the problem (optional, default "Unknown")
`license`	License (optional, default "cc by-sa"
`rights_owner`	Who owns the rights to the problem (mandatory)
`keywords`	Some keywords describing the problem (optional)
`difficulty`	Guesstimated difficulty (optional)
`limits`	Configuration of limits (optional, see below for details)
`validator`	Output validator configuration (see Section 7.1, optional)

For an explanation of the cc by-sa license, see
http://creativecommons.org/licenses/by-sa/3.0/.

2.1 Configuration of Limits

The limit configuration in problem.yaml is a sequence of mappings with the following keys:

Key	Description
`time_multiplier`	See Section 5.1 (optional, defaults to 5)
`time_safety_margin`	See Section 5.1 (optional, defaults to 2)
`memory`	Memory limit in MB (optional, default is 2048)
`output`	Output limit in MB (optional, default is 8)
`compilation_time`	Compilation time limit in seconds (optional, default is 60)
`validation_time`	Output validator time limit in seconds (optional, default is 60)
`validation_memory`	Output validator memory limit in MB (optional, default is 2048)
`validation_output`	Output validator output limit in MB (optional, default is 8)

```
# Problem configuration
source: ICPC Mid-Atlantic Regional Contest
author: John von Judge
license: cc by-sa
rights_owner: ICPC

limits:
   time_multiplier: 5
   time_safety_margin: 2
   memory: 4096
   output: 16
   compilation_time: 240
   validation_time: 240
   validation_memory: 3072
   validation_output: 4

validator: space_change_sensitive float_absolute_tolerance 1e-6
```

Fig. 2: A maximal problem.yaml, specifying everything explicitly

3 problem_statement/ – Problem Statements

This directory contains problem statements, in LaTeX, one or more languages. Typically it will just contain a file `problem.tex` which contains a problem statement in English.

The LaTeX files should contain the problem text itself, including input and output specifications, but not sample input and output. They should reference auxiliary files (e.g., images) as if the working directory is `<shortname>/problem_statement/`.

3.1 Language Support

Problem statements in other languages are provided in files named `problem.<language>.tex`, where `<language>` is an ISO 639-1 alpha-2 language code (e.g., "en" for English or "sv" for Swedish). For English, either `problem.tex` or `problem.en.tex` can be used (but both are not allowed to exist).

3.2 Writing Problem Statements

Problem statements are written in LaTeX, containing only the actual problem statement, no headers or sample data (the .tex file is included in a template that adds these). A typical problem.tex looks as follows:

```
\problemname{Name of Problem}
```

```
<Problem statement here>

\section*{Input}

<Input description here>

\section*{Output}

<Output description here>
```

3.2.1 Special Commands

The problem name is expected to not contain any LaTeX special characters. In case it does, say if the problem name is $x^2 + y^2 = z^2$, a "plain" version of the problem name must be specified by writing

```
%% plainproblemname: x^2 + y^2 = z^2
```

3.2.2 General Guidelines

Try to keep your LaTeX code as clean as possible, avoiding contorted tweaks to get your problem to look like you want. It should be possible to convert the problem statement to both pdf and html reliably.

3.2.3 Images

The graphics file formats supported are .jpg, .jpeg, .png, .gif, and .pdf. When including a file, say, "snarks.png", you can omit the extension and just write "snarks", and let the system find the appropriate file for you.

We distinguish between two types of pictures used in problem statements: illustrations and figures.

An *illustration* is a non-essential picture whose only purpose is to make the problem statement look prettier. The template provides a command
`\illustration{width%}{image}{attribution}`
to typeset illustrations. Its arguments are:

- `width%`: a number between 0 and 1, the desired width of the illustration, as fraction of the total page width.
- `image`: the image file to be included.
- `attribution`: attribution for the image.

Here, attribution is the source of the image, if it needs to be specified. For instance, if the image is a picture from Flickr (with appropriate license), the attribution could be:

```
Photo by \href{http://www.flickr.com/photos/stewart/3127046103/}{Stewart}
from Flickr
```

A *figure* is an essential picture explaining or clarifying some part of the problem statement. Figures should be typeset using the standard LaTeX figure environment and `\includegraphics` (and possibly the `\subfigure` command). A typical use would be:

```
\begin{figure}[!h]
\includegraphics[width=0.5\textwidth]{snarks}
\caption{The seven different snarks}
\end{figure}
```

3.2.4 Sample Data

Sample data will be taken from `data/sample/`. All pairs of matching (`.in`, `.ans`) files there will be used as sample data, in alphabetical order by file name.

3.2.5 Available Packages

The standard template in which problem.tex is used loads the following useful packages that you may or may not need.

- amsmath
- amssymb
- graphicx
- subfigure
- wrapfig
- listings
- fancyvrb
- url
- fontenc[OT2,T1]
- inputenc[utf8]
- ulem[normalem]

In addition, the template loads several packages providing functionality that problem.tex should typically not bother with. For reference, these packages are:

- times
- fancyhdr
- import
- geometry[left=1in,right=1in,top=0.75in,bottom=0.75in]

- hyperref[colorlinks=true,implicit=false]

If there is some additional package that you feel is essential for your problem statement you should perform the following steps:

1. Reconsider if it is really essential.
2. Mail `austrin@scrool.se` to discuss inclusion of the package in the template.

4 data/ – Test Data

The test data are provided in subdirectories of `<short_name>/data/`. The sample data is in `<short_name>/data/sample/` and the secret test data in `<short_name>/data/secret/`.

All input and answer files have the filename extension, .in and .ans, respectively. Every .in file present must have a matching .ans file, and every .ans file must have a matching .in file.

Optionally a text file (with filename extension .txt) describing the purpose of an input file may be present. For each .txt file there must be a matching (.in,.ans) pair.

5 submissions/ – correct and incorrect solutions

The submission directory contains various sample solutions. This is further subdivided into four directories `accepted`, `wrong_answer`, `time_limit_exceeded`, `run_time_error`, based on how the code is expected to be judged.

It is mandatory to supply at least one `accepted` solution. It is in general recommended to supply at least one `wrong_answer` and `time_limit_exceeded` solution, but their usefulness depends on the specific problem.

5.1 Time Limits

The time limits are determined as follows. First, all the provided `accepted/` solutions are run, once for each input file. Let $t_{\max}$ be the maximum running time among all these runs. The time limit is then set to be $t_{\lim} = \lceil t_{\max} \cdot M \rceil$ where M is the value of the `time_multiplier` parameter from the limits configuration of problem.yaml (See Section 2.1).

Furthermore, it is expected that all of the provided `time_limit_exceeded` submissions run for at least $t_{\lim} \cdot S$ seconds, where S is the value of the `time_safety_margin` parameter from the limits configuration.

Some "best practices" to help guide your problem creation are as follows:

- The larger the `time_safety_margin`, the better. If you need to make it smaller than 2 in order for the `time_limit_exceeded` solutions to time out, your problem has issues.
- Time limits should typically be on the order of a few seconds.

6 input_format_validators/

An input format validator is a program that checks that input files adhere to the problem specifications.

The input format validator should read an input file on stdin. If the input file satisfies the specifications, the validator exits with exit code 42. Any other behaviour from the validator (such as a crash, or exiting with exit code 0), means the input file was not accepted.

Input format validators should be as pedantic as possible. In particular, they should complain if the input file contains spurious spacing, or if the input file contains extra data after the last test data (e.g., if the input file is supposed to contain a line with three integers but actually contains two such lines, or even just an empty second line, that should be detected).

It is usually convenient to write these in Python using regular expressions. See the input format validator for the `different` problem for an example.

If there is more than one input format validator, an input file is considered OK if *all* the input format validators accept it, and rejected otherwise.

The input format validator can print extra information to stdout and stderr at its leisure.

Unlike output validators, `problem.yaml` has no effect on how (or whether) input format validators are run.

7 output_validators/

An output validator checks whether the output of a submitted solution to the problem is correct. There is a relatively versatile default validator available that is sufficient for most problems.

The validator choice is configured by the `validator` field in problem.yaml (see Section 2). This field is essentially just a list of arguments that are passed on to the validator program. If the first word of the validator field in problem.yaml is "custom", it indicates that the problem needs a custom output validator and that the default output validator should not be used. The remainder of the validator field is passed as command-line arguments to the custom validator (to all of them, if several are provided).

If the first word of the validator field is *not* "custom" (or if the validator field is not specified), the default validator is used, and the entire validator field is passed as command-line arguments to the default validator. In this case, the `output_validators` directory *must not* exist.

If there is more than one output validator, an output is considered OK if *all* the output validators accept it, and rejected otherwise.

7.1 Default validator options

The default validator is essentially a beefed-up diff. In its default mode, it tokenizes the files to compare and compares token by token. It supports the following command-line arguments to control how tokens are compared.

Argument	Description
`case_sensitive`	indicates that comparisons should be case-sensitive
`space_change_sensitive`	indicates that changes in the amount of whitespace should be rejected (the default is that any sequence of 1 or more whitespace characters are equivalent).
`float_relative_tolerance` ε	indicates that floating-point tokens should be accepted if they are within relative error $\leq \varepsilon$
`float_absolute_tolerance` ε	indicates that floating-point tokens should be accepted if they are within absolute error $\leq \varepsilon$
`float_tolerance` ε	short-hand for applying ε as both relative and absolute tolerance.

Note that when supplying both a relative and an absolute tolerance, the semantics are that a token is accepted if it is within *either* of the two tolerances.

When a floating-point tolerance has been set, any valid formatting of floating point numbers is accepted for floating point tokens. So for instance if a token in the answer file says `0.0314`, a token of `3.14000000e-2` in the output file would be accepted. If no floating point tolerance has been set, floating point tokens are treated just like any other token and has to match exactly.

7.2 Writing custom validators

Not documented yet. See the example validator in the `different` example package.