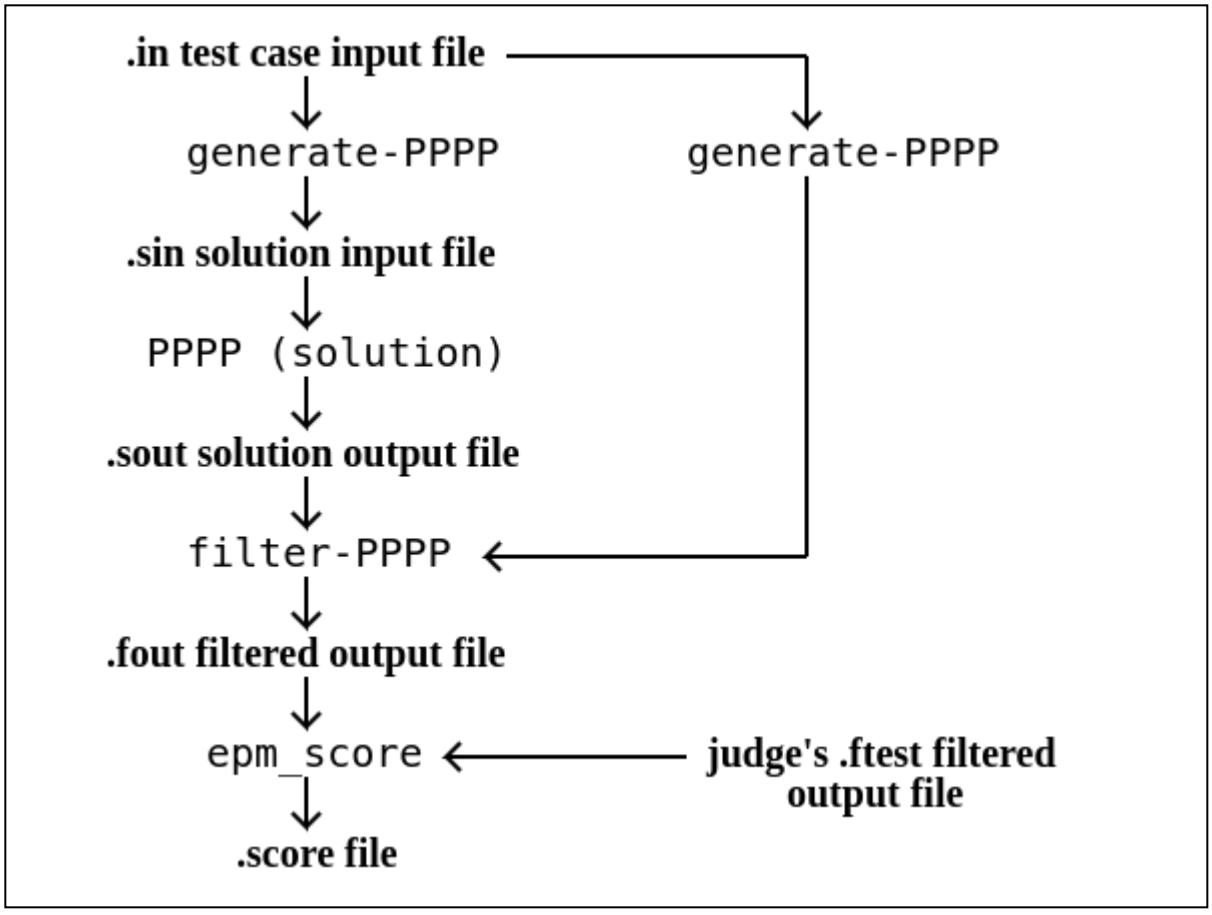
EPM Guide

Introductory Note

EPM (the Educational Problem Manager) is intended to make it easier for one user to develop a programming problem and then make it available for other users to solve. Problem development is mostly a matter of creating test case input and output in a form suitable for automatic scoring, and then debugging the problem solution and specification against the test cases.

The following diagram specifies the data flow used to score a test case for a problem named PPPP. For simple problems, generate-PPPP simply copies its input to its output removing comments, and filter-PPPP simply copies its input to its output (epm_score ignores comments). In more complex cases generate-PPPP uses commands from the .in file to generate .sin solution input using a pseudo-random number generator, and filter-PPPP analyzes the .sin and .sout files to produce a summary used in scoring (e.g., it might read a graph from .sin and check that a path in .sout is in fact a path in the graph, outputting the path length if it is and an error message otherwise).



Getting Started

To open an account, all you need is an e-mail address. Open a tab using the URL of the EPM server you want to access, and enter your e-mail address.

You will be e-mailed a confirmation number which you must enter into your browser. An auto-login period of 2 days is started when you enter the confirmation number and provide the additional information required to establish a new account. At the end of each auto-login period another confirmation number is sent when you next log in. The first auto-log period is 2 days, the second 7 days, and all others 30 days.

Note that most of the additional information will be visible to other users of the EPM server. The exception is the non-domain part of your e-mail addresses, which will NOT be visible.

After you have logged in, go to the Project Page. When you log in the next time, you will be automatically sent to the Project Page.

The buttons marked ? access the Help Page, which has detailed documentation for various pages and parts of pages. **DO NOT USE** the browser Back or Forward buttons, as these will invalidate your tab. You may use the refresh keys, or the ↻ refresh button next to the ? help button. Many pages can be put into an editing mode in which the ↻ button disappears and the refresh keys are disabled.

Solving a Problem

To solve a problem, you first want to `pull' the problem from a project. Go to the Project Page and select the problem list you want to pull the problem from. For example, select `demos Problems'. Then click the `Pull From Project' button.

Next click the oval of the problem you want to pull. For example, click the `reverser' oval, which will turn black. Next click the `Submit' button. The black oval should turn green indicating success, and `Done!' should appear.

After seeing `Done!', click on `Finish'. Then in the `Create Tab for' problem selector, select `reverser'. A new `reverser Problem Page' tab will be created. You now have two tabs, the one just created and the original `Project Page' tab.

Go to the `reverser Problem Page' tab and click on `reverser.pdf' to see the problem specification.

You can write your own solution, or you can click on `View Downloads' to get a window with demos problems solutions of various kinds. You need to get a solution on your own computer, at which point you use `Upload a File' to upload the solution to your reverser problem. When it is uploaded it is compiled, and if there are compilation errors, error messages will be displayed and the solution will NOT be entered into your EPM problem directory. If there are no errors, your solution and its executable file will be entered into your problem directory, or `kept'.

You can see more detail about what happened during the compilation by clicking the downward arrow buttons on the `Commands Last Executed' and `Working Files ...' sections. Working files are files output by the commands that were NOT kept, and are more likely to be of interest when the commands fail and there are no kept files.

Each file in your problem directory has a line in the `Current Problem Files' section of the Problem Page, and each working file from the last executed command has a line in the `Working Files ...' section. After a set of commands execute, one or two files, either kept or working, are high-lighted, and may be automatically shown.

Lines for files with names of the form `00-*-reverser.in' are inputs for sample test cases. If you click on the `=>.score' button for such a line, the score of the test case will be computed in the `00-*-reverser.score' file.

You can show any of your problem directory files or working files that are ASCII text or pdf by clicking either the file name if it is a button, or the downward arrow after the file name. An exception is a very short one-line file whose entire contents is displayed in { } brackets at the end of the file's Problem Page line.

If there are no errors in an operation, only the kept files are high-lighted by making their lines dark-tan instead of light-tan. These files are always in the `Current Problem Files' section, and except for PDF files, are usually not automatically shown.

If there are errors in an operation, no files are kept, and some of the working files are high-lighted with yellow. These yellow high-lighted files are the files you should look at to see what the errors were. Often these files are automatically shown.

In general the operations you perform consist of either uploading a file or making one file from another file. When making one file from another, and the two file names are identical but for their extensions. In addition to making the .score file you can make files with the following extensions:

.sin solution input file
actual input to your solution program
made by filtering .in file through the generate-* program

.sout solution output file
actual output from your solution program

.fout filtered solution output file
made by filtering .sout and .sin files through the filter-* program
compared with judge supplied .ftest file to compute .score file

.dout debugging output
made by inputting .sin file to your solution
running under a debugger to find lines
that segment fault or are in an infinite loop

For the `reverser' problem the .sin file is identical to the .in file with comments removed, and the .fout file is identical to the .sout file.

The best way to debug your program is to put statements in your code that output extra information. Any output line that begins with `!!**' is treated as a comment line and ignored, so you can output as much debugging information as you like in these lines. Debugging comment lines will appear in the .sout file and .fout file and will be ignored by the epm_score program that compares the .fout and .ftest files.

On the other hand if your program blows up or runs out of CPU time, and you want to know the line number of the statement executing when your program failed, you can make a .dout file.

After the scores for the sample test cases are all `Completely Correct', the problem can be submitted by going to the `Run Page'. On this page if you click on the `Run' button of the `sample-reverser.run' line all the sample test cases will be run and the result will be placed in the `sample-reverser.rout' (run output) file. If instead you click on the `Submit' button on the `submit-reverser.run' line, all the test cases, including the hidden judge' test cases with names of the form `01-*-reverser.in', will be run and the total result will be displayed in the `submit-reverser.rout' file.

If there is an error in a test case during a `Submit', the test case .in and .ftest files will be linked into your Problems Page so you can debug the test case as you would a sample test case.

If you have downloaded a solution from the `Downloads Page', you may wish to use it to see what happens when a solution has errors. If you look at a `reverser' solution, you will see that it has a section where different types of errors can be easily introduced either by defining a single macro or by changing the value of a single variable. If you use this to introduce an error in the solution, you can then try using the erroneous solution to produce `.score' files on the `reverser Problem Page' or `.rout' files on the `reverser Run Page'.

More detailed information is available in the Help Page accessed by the `?' buttons.

Creating a Problem

To create a problem you start with a problem name. Try to choose a name that is not, and likly will not, be the name of some project problem, as your problem name will become embedded in all your problem file names. We will let PPPP stand for the problem name in what follows.

Then create the problem on the Projects Page by typing its name into the `New Problem Name' box. This will create a tab for the new problem.

A problem needs first a solution, that is, a program written in C, C++, JAVA, or PYTHON that will solve the problem. It is also well to start with a problem specification written in LATEX.

From the new problem tab you can go to the `Downloads' Page. There you will find sample solutions, a sample problem specification, and a template for writing problem specifications in LATEX.

When you have written your PPPP.c, PPPP.cc, PPPP.java, or PPPP.py solution file, upload it into the new problem tab, and it will compile. Debug its compilation until there are no compilation errors.

Similarly write a LATEX PPPP.tex problem specification file and upload it. It will be compiled by pdflatex. Debug it until it has no errors.

The next step is to make a sample input file named `00-000-PPPP.in'. Upload it, and it will be run through your solution to produce the `00-000-PPPP.sout' solution output file. If there are no detected errors in this, the .in and .sout files will be kept.

You should write your solution so it checks its input for formatting and number-out-of-bounds errors, and if it finds some, either halts (as with assert statements) or writes messages to the standard error output. Other solutions will do not do this, but your solution is a judge's solution, and some code somewhere must check the input for formatting errors. If you do this, then when `00-000-PPPP.in' uploads with no errors, you will know it has the correct format and its numbers are within the proscribed ranges.

Note that for the moment we are ignoring the possibility of having special generate-PPPP and filter-PPPP programs for the problem (see the diagram in the Introductory Note). The system will automatically supply default versions of these which just copy their input to their output (though the default generate-PPPP removes comments).

Read your `00-000-PPPP.sout' file and see if it is correct. So how do you know it is correct? Well, the best way is to put debugging statements in your code that output extra information. Any output line that begins with `!!**' is treated as a comment line and ignored, so you can output as much debugging information as you like in these lines. These debugging comment lines will appear in the .sout file and .fout file and will be ignored by the epm_score program that compares the .fout and .ftest files.

When you think `00-000-PPPP.sout' is correct, on its line click `=>.fout' to make the filtered output file `00-000-PPPP.fout', and then on the line for that file click on `=>.ftest' to make the `00-000-PPPP.ftest' file that is the judge's test data. The .ftest file is made by simply copying the .fout file.

At this point you can click the `=>.score' button on the `00-000-PPPP.in' file and it will score your sample input. Of course the score will necessarily be `Completely Correct', since your .ftest file is a copy of your .fout file, and the score is computed by comparing these two files.

Now the files in your problem will be as follows:

Files That Were Uploaded:
PPPP.tex
PPPP.c, PPPP.cc, PPPP.py, or PPPP.java
00-000-PPPP.in

Files Made During Upload:
PPPP.pdf (on uploading PPPP.tex)
PPPP, PPPP.pyc, or PPPP.jar (on uploading PPPP.c, etc.)
00-000-PPPP.sout (on uploading 00-000-PPPP.in)

Files Made From Other Files:
00-000-PPPP.fout (made from 00-000-PPPP.sout)
00-000-PPPP.ftest (made from 00-000-PPPP.fout)
00-000-PPPP.score (made from 00-000-PPPP.in)

Next you should proceed to make more sample input files with names of the form `00-001-PPPP.in', `00-002-PPPP.in', etc. The `00-' at the beginning of the file name specifies that it is sample input that will be given to all users who try to solve the problem. Names beginning `01-', `02-', etc are judging inputs that are hidden from users who are trying to solve the problem.

When you have all your sample inputs, make a `sample-PPPP.run' file that just lists the names of your .in sample input files, one name per line. Then upload it. Upon uploading it will be checked to be sure that all the .in input files it names exist and all their associated .ftest files exist.

On the `sample-PPPP.run' line there is a `Run' button. If you click it, the tab will switch to the Run Page and the .run file will be run. The results will be displayed in a `sample-PPPP.rout' file.

Assuming you made only one additional sample .in file, at this point the files in your problem will be as follows:

Files That Were Uploaded:
PPPP.tex
PPPP.c, PPPP.cc, PPPP.py, or PPPP.java
00-000-PPPP.in
00-001-PPPP.in
sample-PPPP.run

Files Made During Upload:
PPPP.pdf (on uploading PPPP.tex)
PPPP, PPPP.pyc, or PPPP.jar (on uploading PPPP.c, etc.)
00-000-PPPP.sout (on uploading 00-000-PPPP.in)
00-001-PPPP.sout (on uploading 00-001-PPPP.in)

Files Made From Other Files:
00-000-PPPP.fout (made from 00-000-PPPP.sout)
00-001-PPPP.fout (made from 00-001-PPPP.sout)
00-000-PPPP.ftest (made from 00-000-PPPP.fout)
00-001-PPPP.ftest (made from 00-001-PPPP.fout)
00-000-PPPP.score (made from 00-000-PPPP.in)
00-001-PPPP.score (made from 00-001-PPPP.in)
sample-PPPP.rout (made from sample-PPPP.run)

Next make the judge's hidden data `01-000-PPPP.in', etc. Sometimes judge's data has larger test cases that must be generated randomly. To do this, you must write a `generate-PPPP' program in C or C++. There is a template

and examples on the `Downloads Page'. If you look at other problems that have non-default generate programs, you can see what they do by clicking the `=>.txt' button on their `generate-...' lines.

Sometimes there may be more than one correct solution output and you may need to write a `filter-PPPP' program in in C or C++ to check the output. For example, if the solution is supposed to be a shortest path through a graph, the filter program may read the graph from the .sin file and the solution from the .sout file, check that the solution path is actually a graph path, and if yes output the path length, but otherwise output error messages. Then if the .sout file contains only actual paths, it and the .ftest file will contain only path lengths, and these may be compared to see if they match.

There is a `filter-PPPP' source template and an example on the `Downloads Page', and you can see what other problems do by clicking the `=>.txt' button on their `filter-...' lines.

After you have finished making judge's .in and .ftest files, make a `submit-PPPP.run' file that lists ALL the .in files, both sample and judge's hidden. Test this last using its `Run' button.

To get `submit-PPPP.run' to work, you may need to adjust the problem options on the problem Option Page. For example, you may need to increase the allowed CPU time. Be sure to allow more time for JAVA and PYTHON solutions than for C/C++ solutions. For example, you may wish to use the ratios 1:2:3 for C/C++:JAVA:PYTHON times.

At this point the following summarizes the files in your problem:

Files That Were Uploaded:

PPPP.tex
PPPP.c, PPPP.cc, PPPP.py, or PPPP.java
generate-PPPP.c or generate-PPPP.cc (if needed)
filter-PPPP.c or filter-PPPP.cc (if needed)
00-000-PPPP.in
.....
sample-PPPP.run
submit-PPPP.run

Files Made By Clicks:

00-000-PPPP.fout
.....
00-000-PPPP.ftest
.....
00-000-PPPP.score
.....
sample-PPPP.rout (by clicking Run button)
submit-PPPP.rout (by clicking Run button)

Once you have debugged `submit-PPPP.run' and have a good looking problem specification in `PPPP.pdf', you can push the problem to a project and see if others will try to solve it. To see how to push the problem, go to the Help Page, click on the Table of Contents down-arrow, and click on Pushing Problems under Project Page.

When you push the problem the following files will be copied to the project problem directory so that others can use them:

PPPP.pdf
generate-PPPP
filter-PPPP
00-000-PPPP.in
.....

00-000-PPPP.ftest
.....
sample-PPPP.run
submit-PPPP.run
PPPP.optn (invisible file containing options that you changed)

In addition the following source files are saved in a subdirectory of the project problem directory so they will be available to maintain the project problem in the future:

PPPP.tex
PPPP.c, PPPP.cc, PPPP.py, or PPPP.java
generate-PPPP.c or generate-PPPP.cc (if uploaded)
filter-PPPP.c or filter-PPPP.cc (if uploaded)