Problem Setting Project

CSCI-UA.0480-004
v1.02

April 18, 2018

1 Introduction

In this project you will develop an algorithmic problem that challenges your classmates. You will experience the process of designing, developing and testing an algorithmic problem.

2 Team

The project is carried out in teams of three. Please find your teammates as soon as possible so that you can start developing your problem idea.

It is fully up to you to decide how to split the work among the team members. A typical collaboration between problem developers is:

- A problem setter writes the problem statement, designs the test data, provides an AC solution.
- One or two problem testers write alternative AC solutions, check the strength of the test data, provide anti-solutions and hacky solutions to ensure data quality.

Note that a problem setter's duty is heavier than a problem setter. So you may want to balance by having more people doing the setter's role.

* We do not accept teams with fewer than three. You may post on Piazza to search for teammates. The number of enrolled students is divisible by three so everyone should be able to team up.

3 Timeline

- **Sunday, April 22**: Problem idea submission. By this date you will submit your problem idea. We will check and confirm your problem before you develop the problem package. You are encouraged to submit your problem idea early so as to have more time to develop the problem package.

- **Sunday, April 29**: Problem package submission. By this date you must submit a usable problem package. See the problem package section for the content of a package.
- **Monday, April 30 – Wednesday, May 9**: Problem solving. You will have about 10 days to solve your classmates' challenges as a warmup for the final exam!

4 Submission

All submissions for the problem setting project must be uploaded on NYUClasses. Two entries will be created for idea proposal and problem package separately. Note that we will immediately begin checking your submission once you upload your file. Please only submit when you are ready. Each team only need to submit once via any team member's account.

All PDFs in this project should be compiled from the provided Tex template. You may edit and compile a tex file using local tex installation (e.g. texlive), or using an online tex tool (e.g. sharelatex).

Problem package should be submitted as a single zip file. The content of the zip file should contain everything listed in Section 6.

5 Problem Idea

Please submit on NYUClasses a PDF that briefly describes your intended problem. A template (idea_proposal.tex) has been provided for you to write the problem idea. Please complete each section in the template to propose your problem.

5.1 A Good Problem Idea

A good problem should present an interesting algorithmic concept. It should be difficult enough to offer a challenge, but not overwhelmingly difficult. We ask that you choose a problem that is algorithmically challenging, instead of implementation-wise challenging.

Ideally, you would like a new idea (all contest problems require new ideas). Since this is a intro-level course, it is acceptable if your idea collides with some existing problems. But we ask that you do some search for similar problems before you propose your idea. If you do find a highly similar problem, please try to use a different idea or extend your idea so that it differs from the existing problem. We understand that some overlap may be unavoidable, so please do it to your best knowledge. At the very least, you may want to avoid using an idea from "the top 10 interview questions".

5.2 Algorithm Topics

Your problem must be solvable by an algorithm introduced or to be introduced in this course according to the syllabus. The expected solution should not use an algorithm or advanced technique that exceeds the scope of this course.

6 Problem Package

A problem package consists of several components:

- problem_statement
- submissions
- input_format_validators
- data
- gen
- problem.yaml

A sample problem package has been provided in skeleton.zip. This is the package for the Simple Max question we saw in HW1.

* Everything inside the submitted problem package must be work of your team. You should never take any statement/submissions/data from existing problems.

6.1 Kattis Problemtools

We use the Kattis problemtools to develop and test the problems. You can find more info about the problemtools here. First you need to have an environment that has problemtools installed and is able to run `verifyproblem`. For the skeleton package, the command `verifyproblem skeleton` should execute successfully with zero warnings and errors.

You need an Ubuntu environment to run `verifyproblem`. If you do not have an Ubuntu machine, it is recommended that you run the tool in a docker container or a virtual machine. Here we provide instructions on how to run `verifyproblem` from Coda's docker container. If you would like to run it on an Ubuntu system you have on your machine or some VM, you may follow the problemtool's installation instruction on GitHub.

Problemtools in a Docker Container

First install docker on your machine. Instructions on docker installation can be found here. Then pull and launch the container:

```
docker pull szfck/nyu-problemtools:latest

docker run -dit --name problemtools -v \
  {your_problem_package_directory}:/usr/share/src \
  szfck/nyu-problemtools:latest

docker exec -it problemtools /bin/bash
```

The first command pulls the docker image (it may take a while to finish). The second command creates a container called "problemtools" with the image, and mounts your local problem package directory (the directory that contains the folder "skeleton") to `/usr/share/src` within the container's file system. Please make sure that your local directory is mountable to docker. You can check the mounting accessibility in the docker menu: preferences – file sharing. Note that the second command is multi-line. The backslash may not be copied correctly from a PDF and you may need to copy line-by-line or type it yourself. The last command enters the container with a bash prompt.

Now you can run `verifyproblem` in the container:

```
cd /usr/share/src

verifyproblem skeleton
```

First go to the mounted directory that contains the problem package. Then you can run `verifyproblem` on the given problem package. You should see that `verifyproblem` runs through all submissions and reports no errors and warnings.

6.2 Problem Statement

This is a tex file giving the description of the problem. Please follow the given format in "problem.en.tex" to write the problem description. In the first section under "problemname", write a few paragraphs for the story and task. In the input section, specify how the input is provided. In the output section, specify what is to be printed.

A problem statement is preferably concise. Fun stories can be used to make the problem more intriguing, but the story should not unnecessarily complicate the reader's understanding of the problem. Typically, the statement should have at most three paragraphs.

Please use *math* style for numbers, variables, and formulae. Every variable must have its range defined in the input section. Typically the range immediately follows the variable in brackets, e.g. "There is a single integer x $(1 \leq x \leq 10)$ on the first line.". All variables should use lowercase letters (e.g. n and m) instead of uppercase letters (N and M), unless the problem requires a distinction between lowercase and uppercase variables.

The problem statement can be compiled into PDF using `problem2pdf skeleton` within the docker container.

* Your problem shall not contain subtasks.

6.3 Submissions

This contains several folders named by their expected verdicts. All submissions in the "accepted" folder are expected to pass, and all submissions in the "wrong_answer" folder are expected to get wrong answer, and so on. When `verifyproblem` is run, it complains if some of those solutions do not get their expected verdicts.

A well-designed problem may have multiple expected solutions from the problem setter and the problem tester. Please provide at least two accepted solutions written independently, and preferably in both C++ and Java. If the problem has different types of solutions (different algorithms), you may want to prepare all these types of solutions.

A well-designed problem may also have several anti-solutions the author intends to fail. For example, if there exists an incorrect greedy solution the author may write such a greedy solution and let `verifyproblem` confirm that it is indeed caught by one of the tests.

6.4 Input Format Validator

The purpose of an input format validator is to ensure that the test cases of the problem strictly follow what is described in the problem statement. It also ensures that the data are formatted correctly.

An input format validator must be strict and thorough by all means. It should check the value of every token to make sure that it is valid. For example, if the problem states that the integers are between -100 and 100, then it is up to the input format validator to compare each integer against this range. If the problem states that a given graph is connected, then the input validator should also check the number of connected component in the input to see if it is actually a single component.

The validator must also prevent other format errors such as trailing spaces on a line, which will break an innocent program that splits by spaces. Other format errors include extra spaces between tokens, tabs in place of spaces, leading zeroes in numbers, etc.

An input format validator must return 42 when the check passes. Any other return code indicates error.

You may write an input format validator in any language of your choice as long as it is supported by the verifyproblem tool. You can also write multiple validators, e.g. one validator for checking value ranges and another validator for checking graph connectivity.

6.5 Data

The data folder has two sub-folders: sample and secret. The sample folder contains the sample cases that are visible to the problem solvers. The secret folder contains the secret test data that is actually executed by the judge machine. Each test file has the suffix ".in" and its corresponding answer file must be given a same name with suffix ".ans", e.g. "tree.ans" is the answer to "tree.in". The names of the test cases do not necessarily need to be numbered like 001, 002, etc. But the cases will be tested lexicographically based on their file names, so it's best practice to number them in the intended execution order for better readability.

Each test case should be meaningful and provide unique coverage. It is okay to generate 3 to 5 random cases, but it is a bad idea to generate tens of cases with a same random pattern. Some cases may actually take quite some efforts to construct, e.g. corner case, greedy breaker, degenerate case, etc.

Typically a test file should not exceed 2–3MB in size. In other words, it should contain at most $10^5 - 10^6$ tokens. Try to avoid excessively large cases. You can actually enforce an $O(nlogn)$ solution by 10^5-scale input.

We recommend developing a problem that has fixed answers. That is, the problem does not need to have a special program (called "output validator") to allow multiple possible answers. If you do have a strong reason to allow multiple answers, please contact us for assistance.

6.6 Generator

A generator script is used to generate the test cases for the problem. This is likely the part of the package that needs the most work. Sometimes it is possible that a problem has a neat and short solution, but to thoroughly test a solution its generator has hundreds of lines.

You are free to use any language other than Python for a generator. If your test data includes a test case that is generated by script, you must include the generator script in your problem package. We kindly ask you to comment the generator code so that we can understand how the tests are generated. You can have multiple script files to generate different types of test input.

You can also generate some of the test cases manually. Typically, you can hard-code them in a text file and then write a wrapper code to copy them into individual test files.

6.7 problem.yaml

This file specifies parameters related to the time limit. Most likely you do not need to touch this file.

"time_multiplier" is a factor for calculating the suggested time limit of the problem. The suggested time limit is set to "slowest AC solution execution time" multiplied by the time_multiplier. For example, if the slowest AC solutions runs for 0.6 seconds, you may see problemtools using a time limit of 2 seconds.

"time_safety_margin" is a factor for calculating the time the TLE solutions take. Every TLE solution should run for more than "time limit" multiplied by this factor. For example, if the time limit is 3 seconds, every TLE solution must run for more than 6 seconds. If a TLE solution runs below 3 seconds, you get a critical error because the TLE solution gets AC. If a TLE solution runs between 3 and 6 seconds, you get a warning because the TLE solution is too close to being accepted. You should design your constraints and test data so that TLE solutions are away from the margin. A problem package should have absolutely zero errors. A problem package should ideally have zero warnings.

If you need floating point tolerance, uncomment the float_tolerance line.

7 Scoring

Your score for the problem setting project is based on:

- Completion (5 points)
- Solutions and anti-solutions (2 points)
- Test data (2 points)
- Level of difficulty (1 point)

7.1 Completion

As long as you submit a verifiable problem package, you will receive full marks for this part. The problem package should not have missing components from those listed in Section 6. When we run `verifyproblem` on your problem, it should report zero errors and warnings.

If your problem contains errors not reported by `verifyproblem`, such as typos in the statement, incorrect formatting of variables, points will also be deducted from this part.

7.2 Solutions and Anti-Solutions

We ask you to thoroughly think about all possible solutions and anti-solutions to your problem. Sometimes a problem may be solved in multiple ways, or a same way but with slight variations. The more different types of solutions you can come up with the better. For anti-solutions, try to think about incorrect greedy solutions, brute-force (timeout) solutions, hacks, etc.

For example, if you neglect an anti-solution or hack, but someone (this can be instructor or graders) finds such an anti-solution after the problem is released, you will lose points on this part.

7.3 Test Data

Your test data should reflect your solution and anti-solution design. If you see that an incorrect greedy should fail, then there must be a test case that breaks it. If you envisions an incorrect solution, but the test cases fail to capture it, this is then a failure on the test data. For example, if you are to block $O(n^2)$ solutions but one of such solutions turned out to pass tightly in the time limit, you will lose points on this part.

Note that if you do not see a possible hack which turns out to pass, this belongs to the previous category and you will lose points on the previous part instead of this part.

7.4 Level of Difficulty

Ideally you would want your problem to be solved by someone, but not everyone.

Your score for this part is $1-4(\frac{x-n/2}{n})^2$, where n is the number of people who attempt the problemset, and x is the number of people who solve your problem.