tcframe Documentation

Release 0.6.0

Ashar Fuadi

February 08, 2016

Contents:

CHAPTER 1

Introduction

tcframe is a framework for generating test cases, designed especially for problems in competitive programming contests. If you are a problem author and want to create test cases for your problem, you are on the right site!

1.1 Philosophy

First, let's begin with a philosophy on why this project is useful.

Why do we need to write generator for test cases?

- Writing test cases manually is error-prone and time-consuming.
- We can easily distribute the test cases, as a single, small file.
- We can easily modify test case values in the future, for example when there are changes in the constraints.

Why do we need a test cases generation framework?

- Not all people know how to write a test cases generator.
- To avoid writing repetitive and boring tasks, for example, creating test case files with appropriate numbering, running official solution against the test case input files, etc.
- To make all problems in a contest have test cases generator with consistent format.

For the above reasons, **tcframe** was developed.

1.2 Features

As a framework, **tcframe** offers the following exciting features:

- Excluding the official solution, we only need to write a single C++ program, called runner program.
- The runner program has a syntax that is easy to read.
- The runner program is nicely structured in several parts, similar to the problem statement: input format, output format, constraints/subtasks, etc.
- The resulting test cases are validated according to the specification constraints/subtasks.
- All errors (for example: constraints unsatisfiability) are presented to the users in human-readable format so that they can easily fix the errors.

1.3 Examples

Here are some example of runner programs written using **tcframe** framework:

1.3.1 For problems without subtasks

```
#include "tcframe/runner.hpp"
using namespace tcframe;

class Problem : public BaseProblem {
protected:
    int A, B;
    int K;

    int result;

    void InputFormat() {
        LINE(A, B);
        LINE(K);
    }

    void OutputFormat() {
        LINE(result);
    }

    void Constraints() {
        CONS(1 <= A && A <= 1000);
        CONS(1 <= B && B <= 1000);
        CONS(0 <= K && K <= 100);
    }
};

class Generator : public BaseGenerator<Problem> {
protected:
    void SampleTestCases() {
        SAMPLE_CASE({
            "1 1",
            "100"
        });
    }

    void TestCases() {
        CASE(A = 1, B = 100, K = 7);
        CASE(A = 100, B = 2000, K = 0);
    }
};

int main(int argc, char* argv[]) {
    Runner<Problem> runner(argc, argv);

    runner.setGenerator(new Generator());
    return runner.run();
}
```

The above runner program, when run, will output:

```
Generating test cases...

[ SAMPLE TEST CASES ]
  problem_sample_1: OK

[ OFFICIAL TEST CASES ]
  problem_1: OK
  problem_2: FAILED
    Description: A = 100, B = 2000, K = 0
    Reasons:
    * Does not satisfy constraints, on:
      - 1 <= B && B <= 1000
```

1.3.2 For problems with subtasks

```
#include "tcframe/runner.hpp"
using namespace tcframe;

class Problem : public BaseProblem {
protected:
    int A, B;
    int K;

    int result;

    void InputFormat() {
        LINE(A, B);
        LINE(K);
    }

    void OutputFormat() {
        LINE(result);
    }

    void Subtask1() {
        CONS(1 <= A && A <= 1000);
        CONS(1 <= B && B <= 1000);
        CONS(0 <= K && K <= 100);
    }

    void Subtask2() {
        CONS(1 <= A && A <= 2000000000);
        CONS(1 <= B && B <= 2000000000);
        CONS(0 <= K && K <= 10000);
    }

    void Subtask3() {
        CONS(A == 0);
        CONS(B == 0);
        CONS(0 <= K && K <= 100);
    }
};

class Generator : public BaseGenerator<Problem> {
protected:
    void SampleTestCases() {
        SAMPLE_CASE({
            "1  1",
            "100"
        }, {1, 2});
    }

    void TestGroup1() {
        assignToSubtasks({1, 2});

        CASE(A = 1, B = 100, K = 7);
        CASE(A = 100, B = 2000, K = 0);
    }

    void TestGroup2() {
        assignToSubtasks({2});

        CASE(A = 1, B = 2, K = 1);
        CASE(A = 0, B = 0, K = 100);
    }
};

int main(int argc, char* argv[]) {
    Runner<Problem> runner(argc, argv);

    runner.setGenerator(new Generator());
    return runner.run();
}
```

The above runner program, when run, will output:

```
Generating test cases...

[ SAMPLE TEST CASES ]
  problem_sample_1: FAILED
    Reasons:
    * Cannot parse for variable `B`. Found: <whitespace>

[ TEST GROUP 1 ]
  problem_1_1: OK
  problem_1_2: FAILED
    Description: A = 100, B = 2000, K = 0
    Reasons:
    * Does not satisfy subtask 1, on constraints:
      - 1 <= B && B <= 1000

[ TEST GROUP 2 ]
  problem_2_1: FAILED
    Description: A = 1, B = 2, K = 1
    Reasons:
    * Satisfies subtask 1 but is not assigned to it
  problem_2_2: FAILED
    Description: A = 0, B = 0, K = 100
    Reasons:
    * Does not satisfy subtask 2, on constraints:
      - 1 <= A && A <= 2000000000
      - 1 <= B && B <= 2000000000
    * Satisfies subtask 3 but is not assigned to it
```

CHAPTER 2

Installation

2.1 Dependencies

tcframe needs the following minimum requirements:

- UNIX-based operating system
- GCC >= 4.7

2.2 Setup

tcframe only consists of C++ header files; nothing to be really installed to your system. Clone the header files from GitHub.

Basic concepts

Let's learn how to write a generator using tcframe with this basic tutorial. Consider the following example problem.

Description

You are given integers N and K, and an array consisting of N integers. Compute the largest possible product of any K elements of the array! Each element can be used more than once.

Input Format

The first line contains two integers N and K. The second line contains N space-separated integers: the elements of the array.

Output Format

A single line containing the required product.

Sample Input

```
5 3
0 1 -1 2 -2
```

Sample Output

```
8
```

Subtask 1:

- N = 1
- 1 <= K <= 100
- -100000 <= A[i] <= 100000

Subtask 2:

- 1 <= N <= 100
- K = 1
- -100000 <= A[i] <= 100000

Subtask 3:

- 1 <= N <= 100
- 1 <= K <= 100
- -100000 <= A[i] <= 100000

We will be writing a test cases generator for this problem. The program we will be writing is called **runner program**. Create a C++ file, for example, **runner.cpp**.

First of all, we need to include the main tcframe runner header file:

```
#include "tcframe/runner.hpp"
using namespace tcframe;
```

Then, we will need to write specifications as will be explained in the next sections.

3.1 Problem specification

The first step is to write a **problem specification** class. It is basically a class that represents a problem's test case variables, I/O format, and the required constraints/subtasks.

Create a class that inherits **BaseProblem**. The convention is to name the class **Problem**:

```
class Problem : public BaseProblem {
protected:

    // components here

};
```

A problem specification consists of several components. Note that publicly defined components must all go to the protected section of the class. You are allowed to write private helper variables and/or methods, of course.

3.2 Problem configuration

This component consists of configuration of several aspects of the problem. To define this component, override the method **void BaseProblem::Config()**. For example, for this problem, we want that the slug (i.e., file prefix) for the test cases to be "k-product".

```
void Config() {
    setSlug("k-product");
}
```

The complete reference of configurable aspects of a problem can be found here: *Problem configuration API reference*.

This component can be omitted if you don't want to override any default settings.

3.3 Input/output variables

Input variables are a collection of variables which compose test cases inputs. They can be usually found in the input format section in the problem statement. For this problem, we have three input variables: **N**, **K**, and **A**. The input variables are defined as protected member variables.

In this problem, we have two scalars (**N**, **K**) and one vector (**A**) as the input variables. We define them as follows:

```
int N;
int K;
vector<int> A;
```

Similarly, **output variables** are a collection of variables which compose test cases inputs. Most of the cases, this is just a single variable and does not have a particular name in the problem statement. Let's just call it result.

```
int N;
int K;
vector<int> A;

int result;
```

The complete reference of input/output variables can be found here: *Input/output variables API reference*.

3.4 Input/output format

Input format specifies how the input variables should be printed in test case input files. To define this component, override the method **void BaseProblem::InputFormat()**. The format is specified in terms of consecutive input **segment**s. Basically an input segment arranges the layout of several input variables.

A test case input file for this problem consists of a single containing **N** and **K**, followed by a single line containing space-separated elements of **A**. We can define that format as follows:

```
void InputFormat() {
    LINE(N, K);
    LINE(A % SIZE(N));
}
```

Similarly, **output format** specifies how the input variables should be printed in test case input files. To define this component, override the method **void BaseProblem::OutputFormat()**.

```
void OutputFormat() {
    LINE(result);
}
```

The complete reference of input/output segments can be found here: *Input/output segments API reference*.

3.5 Constraints

This components specifies the constraints of the problem; i.e., the conditions that must be satisfied by the input/output variables. Two types of problems are supported: the ones without subtasks, and the ones with subtasks.

For problems without subtasks: Override the method **void BaseProblem::Constraints()**.

For problems with subtasks: Override each of the methods **void BaseProblem::SubtaskX()**, where **X** is a positive integer denoting the subtask number.

Note: As of this version, you can define up to 10 subtasks: **Subtask1()** .. **Subtask10()**.

Inside the overriden method(s), we can define the constraints. A constraint is defined with a **CONS()** macro containing a boolean expression.

Let's define the subtasks for this problem.

```
void Subtask1() {
    CONS(N == 1);
    CONS(1 <= K && K <= 100);
    CONS(eachElementBetween(A, -100000, 100000));
}

void Subtask2() {
    CONS(1 <= N && N <= 100);
    CONS(K == 1);
    CONS(eachElementBetween(A, -100000, 100000));
}

void Subtask3() {
    CONS(1 <= N && N <= 100);
    CONS(1 <= K && K <= 100);
    CONS(eachElementBetween(A, -100000, 100000));
}
```

where **eachElementBetween()** is a private helper method, defined as follows:

```
bool eachElementBetween(const vector<int>& A, int lo, int hi) {
    for (int x : A) {
        if (x < lo || x > hi) {
            return false;
        }
    }
    return true;
}
```

Note: As of this version, there is currently no easy way to test a predicate for each element of a vector. The workaround is to write a helper method ourselves, like what we did above.

The complete reference of constraints can be found here: *Constraints API reference*.

We have now completed writing a problem specification class. In summary, our class should look like this:

```
class Problem : public BaseProblem {
protected:
    int N;
    int K;
    vector<int> A;

    int result;

    void Config() {
        setSlug("k-product");
    }

    void InputFormat() {
        LINE(N, K);
        LINE(A % SIZE(N));
    }

    void OutputFormat() {
        LINE(result);
    }

    void Subtask1() {
        CONS(N == 1);
        CONS(1 <= K && K <= 100);
        CONS(eachElementBetween(A, -100000, 100000));
    }

    void Subtask2() {
        CONS(1 <= N && N <= 100);
        CONS(K == 1);
        CONS(eachElementBetween(A, -100000, 100000));
    }

    void Subtask3() {
        CONS(1 <= N && N <= 100);
        CONS(1 <= K && K <= 100);
        CONS(eachElementBetween(A, -100000, 100000));
    }

private:
    bool eachElementBetween(const vector<int>& A, int lo, int hi) {
        for (int x : A) {
            if (x < lo || x > hi) {
                return false;
            }
        }
        return true;
    }
};
```

The nice thing is that this problem specification class is really similar to the problem statement! This class will then serve as a "contract" for the generator, which we will write next.

3.6 Generator specification

The next step is to write a **generator specification** class. It is basically a class that represents a collection of (randomly generated) test cases, based on the specification defined in the problem specification class.

Create a class that inherits **BaseGenerator<T>**, where **T** is the problem specification class. The convention is to name the class **Generator**:

```
class Generator : public BaseGenerator<Problem> {
protected:

    // components here

};
```

Similar to a problem specification, a generator specification consists of several components, which must go to the protected section of the class.

3.7 Generator configuration

This component consists of configuration of several aspects of the problem. To define this component, override the method void BaseGenerator::Config(). Currently, we can define where the test cases are output, and which solution to run on the test case input files.

For this problem:

```
void Config() {
    setTestCasesDir("tc");
    setSolutionCommand("./solution");
}
```

Note: For this tutorial, please create an executable file named "solution" in the same directory as generator.cpp. It could be any solution – for example, a solution that just prints Hello World.

The complete reference of generator configuration can be found here: *Generator configuration API reference*.

The above configuration is the default one. This component can be omitted if you don't want to override any default values, which we will do for this tutorial.

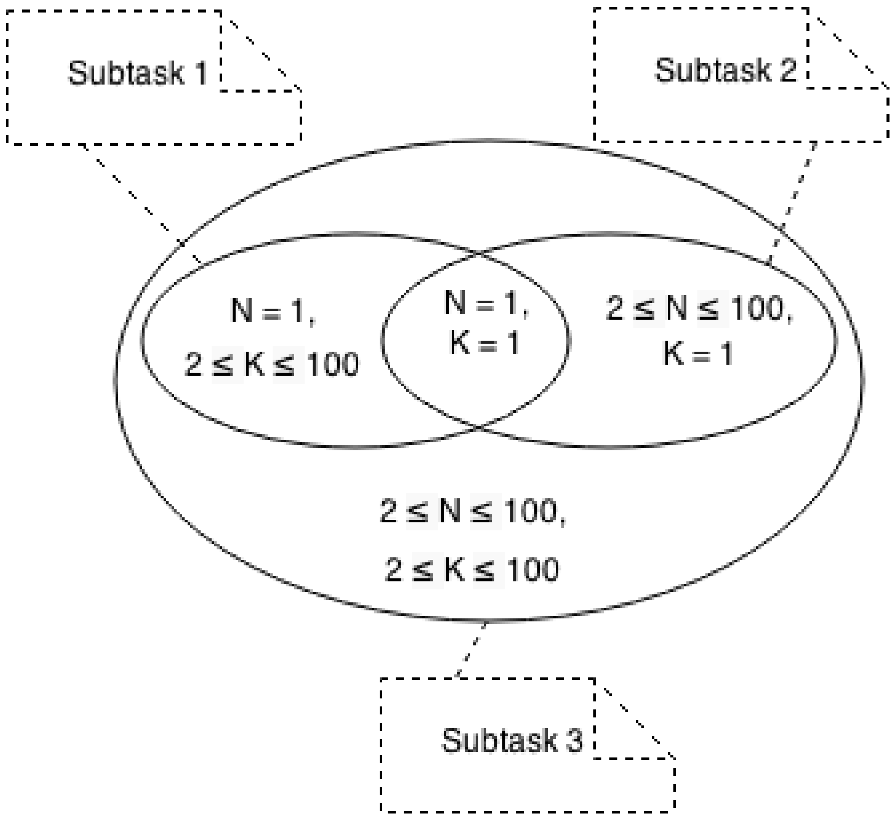
3.8 Test cases

This component specifies a collection of values of the problem's input variables, each of which constitute a test case. Two types of problems are supported: the ones without subtasks, and the ones with subtasks.

For problems without subtasks: Override the method **BaseGenerator::TestCases()**. The content of this method will be explained shortly.

For problems with subtasks: The idea is that a test case must be able to be assigned to more than one subtasks. To support this, we introduce a concept called **test groups**. A test group is a set of test cases that are assigned to the same set of subtasks.

First, create a Venn diagram denoting the valid test cases for all subtasks. For this problem, the diagram will be like this:



In order to have a strong set of test cases, we should create a test group for each **closed region** in the Venn diagram. In this case, we will have four test groups as follows:

- Test group 1: consists of only one test case N = K = 1. Assign it to subtasks {1, 2, 3}.
- Test group 2: generate test cases that satisfy N = 1; 2 <= K <= 100. Assign them to subtasks {1, 3}.
- Test group 3: generate test cases that satisfy 2 <= N <= 100; K = 1. Assign them to subtasks {2, 3}.
- Test group 4: generate test cases that satisfy 2 <= N, K <= 100. Assign them to subtasks {3}.

To define test groups, override each of the methods **BaseGenerator::TestGroupX()**, where **X** is a positive integer denoting the test group number. Then, call **assignToSubtasks(S)** method as the first statement, where **S** is a list of subtask numbers. The remaining content of test group methods are test case definitions which will be explained below.

Note: As of this version, you can define up to 10 test groups: **TestGroup1()** .. **TestGroup10()**.

Inside the methods **TestCases()** or **TestGroupX()**, we can define the test cases. A test case is defined with a **CASE()** macro containing a list of assignment to an input variable or method call. Each CASE() defines a single test case and should assign valid values to all input variables. For example:

```
void TestGroup2() {
    assignToSubtasks({1, 3});

    CASE(N = 1, K = 3, randomArray());
    CASE(N = 1, K = 100, randomArray());
}
```

where **randomArray()** is a private helper method that assign random values (between -100000 and 100000) to each of the element A[0] .. A[N-1]:

```
void randomArray() {
    A.clear(); // important!
    for (int i = 0; i < N; i++) {
        A.push_back(rnd.nextInt(-100000, 100000));
    }
}
```

Note: Yes, we can access the input variables directly inside the generator, even though they belong to the problem specification class!

Here, we are using random number generator object `rnd` that is available inside generator class. The complete reference of randomization methods can be found here: *Random number generator API reference*.

We will also define sample test cases. Each sample test case is independent to each other, and they are not included in any test group. Therefore, for problems with subtasks, we must assign a set of subtasks for each sample test case.

The complete reference of test case and sample test case definitions can be found here: *Test cases API reference*.

3.9 Main function

After writing problem and generator specification classes, write the **main()** function and configure the runner as follows:

```
int main(int argc, char* argv[]) {
    Runner<Problem> runner(argc, argv);

    runner.setGenerator(new Generator());
    return runner.run();
}
```

The complete runner program for this problem is summarized below.

Note that for vector input variables, don't forget to clear them before assigning the values.

```
#include "tcframe/runner.hpp"
using namespace tcframe;

#include <random>
#include <vector>
using namespace std;

class Problem : public BaseProblem {
protected:
    int N;
    int K;
    vector<int> A;

    int result;

    void Config() {
        setSlug("k-product");
    }

    void InputFormat() {
        LINE(N, K);
        LINE(A % SIZE(N));
    }

    void OutputFormat() {
        LINE(result);
    }

    void Subtask1() {
        CONS(N == 1);
        CONS(1 <= K && K <= 100);
        CONS(eachElementBetween(A, -100000, 100000));
    }

    void Subtask2() {
        CONS(1 <= N && N <= 100);
        CONS(K == 1);
        CONS(eachElementBetween(A, -100000, 100000));
    }

    void Subtask3() {
        CONS(1 <= N && N <= 100);
        CONS(1 <= K && K <= 100);
        CONS(eachElementBetween(A, -100000, 100000));
    }

private:
    bool eachElementBetween(const vector<int>& A, int lo, int hi) {
        for (int x : A) {
            if (x < lo || x > hi) {
                return false;
            }
        }
        return true;
    }
};

class Generator : public BaseGenerator<Problem> {
protected:
    void SampleTestCases() {
        SAMPLE_CASE({
            "5 3",
            "0 1 -1 2 -2"
        }, {3});
    }

    void TestGroup1() {
        assignToSubtasks({1, 2, 3});

        CASE(N = 1, K = 1, randomArray());
    }

    void TestGroup2() {
        assignToSubtasks({1, 3});

        CASE(N = 1, K = 2, randomArray());
        CASE(N = 1, K = 10, randomArray());
        CASE(N = 1, K = 100, randomArray());
    }

    void TestGroup3() {
        assignToSubtasks({2, 3});

        CASE(N = 2, K = 1, randomArray());
        CASE(N = 10, K = 1, randomArray());
        CASE(N = 100, K = 1, randomArray());
    }

    void TestGroup4() {
        assignToSubtasks({3});

        CASE(N = 2, K = 2, randomArray());
        CASE(N = 10, K = 10, randomArray());
        CASE(N = 42, K = 58, randomArray());
        CASE(N = 100, K = 100, randomArray());
        CASE(N = 100, K = 100, randomArray());
        CASE(N = 100, K = 100, randomArray());
    }

private:
    void randomArray() {
        A.clear(); // important!
        for (int i = 0; i < N; i++) {
            A.push_back(rnd.nextInt(-100000, 100000));
        }
    }
```

```
};

int main(int argc, char* argv[]) {
    Runner<Problem> runner(argc, argv);

    runner.setGenerator(new Generator());
    return runner.run();
}
```

3.10 Compiling runner program

Suppose that your runner program is **runner.cpp**. Compile it using this compilation command:

```
g++ -I[path to tcframe]/include -std=c++11 -o runner runner.cpp
```

For example:

```
g++ -I/home/fushar/tcframe/include -std=c++11 -o runner runner.cpp
```

Note: The current version needs GCC version >= 4.7.

3.11 Running runner program

Just run

```
./runner
```

There are several command-line options that can be specified when running the runner program. The options mostly override the problem and generator configuration. For example, we can override the specified official solution:

```
./runner --solution-command="java Solution"
```

See *Command-line options* for more information on the command-line options.

The status of the generation of each test case will be output to the standard output. For each successful test case, the input-output file pair will be stored in the specified test cases directory (by default, it is "tc").

Generation can fail due to several reasons:

Invalid input/output format For example: using scalar variable for a grid segment.

Invalid input variable states For example: a grid segment requires that the size is 2 x 3, but after applying the test case definition, the matrix consists of 3 x 4 elements.

Unsatisfied constraints/subtasks The input variables do not conform to the constraints.

CHAPTER 4

Advanced concepts

4.1 Manipulating input variables with different representation

Often, we want to manipulate input variables with a different representation from what is defined in the input format section. For example, suppose that we want to have a tree as an input. In the input format, we specify the tree as a list of edges (U[i], V[i]) as follows:

```
void InputFormat() {
    LINE(N);
    LINES(U, V) % SIZE(N - 1);
}
```

and we want to manipulate the tree as a vector P[], where P[i] is the parent of node i. (I.e., we have private variable vector<int> P in Generator.)

This can be achieved by overriding a special method **BaseGenerator::FinalizeInput()** method and transforming vector P[] into pair of vectors (U[], V[]) in it.

```
void FinalizeInput() {
    U.clear();
    P.clear();
    for (int i = 0; i < N; i++) {
        if (P[i] != -1) {
            U.push_back(i);
            V.push_back(P[i]);
        }
    }
}
```

4.2 Complex input/output format

Suppose we have the following input format description from the problem statement:

> The first line contains an integer N. The next N lines each contains one of the following type:
>
> - 1 A[i] B[i]
> - 2 A[i] B[i] C[i]

How to deal with this one?

Unfortunately, as of this version, the above format is not supported yet. This version only supports input/output formats that are free of custom loops and conditional branches.

I am designing how to support that in the next versions.

4.3 Complex predicates in constraints

Similarly, methods used in defining constraints (**Constraints()**, **SubtaskX()**) must be free of custom loops and conditional branches. If you want to use complex predicate, such as determining whether the input is a tree, create a custom private helper method that return boolean (whether the input is a tree).

4.4 Multiple test cases per file (ICPC-style)

tcframe supports ICPC-style problem as well. In this style, for each of the **SampleTestCases()**, **TestCases()**, and **TestGroupX()** methods, the test cases will be combined into a single file. The file is prepended with a single line containing the number of test cases in it.

To write, an ICPC-style test cases generator, first write the runner as usual, assuming that the problem not ICPC-style. Then, apply the following changes.

4.4.1 Problem specification class

- Declare an integer variable (e.g., **T**) that will hold the number of test cases in a single file.

```
protected:
    int T;

    ...
```

- In the problem configuration, call **setMultipleTestCasesCount()** method with the previous variable as the argument.

```
void Config() {
    ...

    setMultipleTestCasesCount(T);

    ...
}
```

- The input format specification should not contain **T**. It should specify the format of a single test case only. The number of test cases will be automatically prepended in the final combined test case input file.
- We can impose a constraint on **T**, inside **MultipleTestCasesConstraints()** method.

```
void MultipleTestCasesConstraints() {
    CONS(1 <= T <= 20);
}
```

4.4.2 Generation specification class

No changes are necessary.

4.4.3 Solution program

Although the input format only specifies a single test case, the solution should read the number of test cases in the first line. In other words, the solution will read the final combined test cases input file.

Simulating submission

tcframe allows you to simulate solution submission locally, on your machine.

Before simulating a submission, you must have generated the test cases:

```
./runner
```

Then, you can simulate a submission, by executing:

```
./runner submit [<submissionCommand>]
```

where <submissionCommand> is the command for executing the submission. If it is omitted, then the command will be the original solution command used in the generator.

For example, suppose you have written a generator for a problem. Your friend also has written an alternate solution to the problem, and he wants to check whether his solution agrees with yours. Let's assume that his solution file is **alt_solution**.cpp. Compile it into **alt_solution**, place it in the same directory of the runner, and then run

```
./runner submit ./alt_solution
```

The verdict of each test case will be shown. The verdict will be one of the following:

Accepted (AC) The output produced by the submission matches.

Wrong Answer (WA) The output produced by the submission does not match. The diff will be shown, truncated to the first 10 lines.

Runtime Error (RTE) The submission crashed or used memory above the limit, if specified.

Time Limit Exceeded (TLE) The submission did not stop within the time limit, if specified.

The verdict of each subtask will be also shown. The verdict of a subtask is the worst verdict of all verdicts of test cases that are assigned to it. Here, RTE is worse than WA, and WA is worse than AC.

Here is a sample output of a submission simulation for problems with subtasks.

```
Submitting...

[ SAMPLE TEST CASES ]
  k-product_sample_1: Accepted

[ TEST GROUP 1 ]
  k-product_1_1: Accepted

[ TEST GROUP 2 ]
  k-product_2_1: Accepted
  k-product_2_2: Accepted
  k-product_2_3: Accepted

[ TEST GROUP 3 ]
  k-product_3_1: Accepted
  k-product_3_2: Wrong Answer
    * Diff:
(expected) [line 01]    11
(received) [line 01]    12

  k-product_3_3: Accepted

[ TEST GROUP 4 ]
  k-product_4_1: Accepted
  k-product_4_2: Accepted
  k-product_4_3: Accepted
  k-product_4_4: Accepted
  k-product_4_5: Accepted
  k-product_4_6: Runtime Error
    * Execution of submission failed:
      - Exit code: 1
      - Standard error:

[ RESULT ]
  Subtask 1: Accepted
  Subtask 2: Wrong Answer
  Subtask 3: Runtime Error
```

and here is for problems without subtasks

```
Submitting...

[ SAMPLE TEST CASES ]
  k-product_sample_1: Accepted

[ OFFICIAL TEST CASES ]
  k-product_1: Accepted
  k-product_2: Accepted
  k-product_3: Accepted
  k-product_4: Wrong Answer
    * Diff:
(expected) [line 01]    11
(received) [line 01]    12

[ RESULT ]
  Wrong Answer
```

This submission simulation feature is useful for creating "unit tests" for your test cases. For each problem, you can write many solutions with different intended results. For example, solution_123.cpp should pass subtasks 1 - 3; solution_12.cpp should pass subtasks 1 and 2 but not subtask 3, etc.

You can also specify options. See *Command-line options* for available command-line options. The most useful optionr are specifying time and memory limit.

5.1 Brief output

If you want to automate checking the result of each solution, you can set the output of the submission to be "brief", i.e., concise and easy to parse by another program. Just pass the command-line option **--brief**:

```
./runner submit ./alt_solution --brief
```

Here is a sample brief output for problems with subtasks:

```
1:AC
2:WA
3:RTE
```

And here is for problems without subtasks:

```
WA
```

CHAPTER 6

API reference

6.1 Problem configuration

The following methods can be called inside the overridden method **BaseProblem::Config()**.

void **`setSlug`** (string *slug*)

Sets the *slug* of the problem. A slug is a nickname or code for the problem. The produced test case filenames will have the slug as prefix. For example, if the slug is helloworld" then one valid test case filename is "helloworld_1.in".

If not specified, the default slug is "problem".

void **`setTimeLimit`** (int *timeLimitInSeconds*)

Sets the time limit of the problem, in seconds. This time limit is used in simulating submission.

void **`setMemoryLimit`** (int *memoryLimitInMegabytes*)

Sets the memory limit of the problem, in MB. This memory limit is used in simulating submission.

6.2 Input/output variables

There are three types of variables that are supported:

Scalar Variables of built-in integral types (int, long long, char, etc.), built-in floating-point types (float, double), and std::string.

Vector std::vector<**T**>, where **T** is a scalar type as defined above. Note that you cannot use arrays (**T**[]).

Matrix std::vector<std::vector<**T**>>, where T is a scalar type as defined above. Note that you cannot use 2D arrays (**T**[][]).

6.3 Input/output segments

The following macros can be called inside the overridden method **BaseProblem::InputFormat()** or **BaseProblem::OutputFormat()**.

`EMPTY_LINE` ()

Defines an empty line.

`LINE` (*comma-separated elements*)

Defines a single line containing space-separated scalar or vector variables. In case of vector variables, the elements are separated by spaces as well.

element is one of:

- <*scalar variable name*>.
- <*vector variable name*> **% SIZE(**<*number of elements*>**)**. The number of elements can be a constant or a scalar variable.
- <*vector variable name*>. Here, the number of elements is unspecified. This kind of element must occur last in a line segment, if any. Elements will be considered until new line is found.

For example:

```
void InputFormat() {
    LINE(N);
    LINE(A % SIZE(3), B);
    LINE(M, C % SIZE(M));
}
```

With **N** = 2, **A** = {1, 2, 3}, **B** = {100, 200, 300, 400}, **M** = 2, **C** = {7, 8}, the above segments will produce:

```
2
1 2 3 100 200 300 400
2 7 8
```

LINES (*comma-separated vector variable names) % SIZE(number of elements*)
Defines multiple lines, each consisting space-separated elements of given vector variables.

For example:

```
void InputFormat() {
    LINES(V) % SIZE(2);
    LINES(X, Y) % SIZE(N);
}
```

With **V** = {1, 2}, **X** = {100, 110, 120}, **Y** = {200, 210, 220}, **N** = 3, the above segments will produce:

```
1
2
100 200
110 210
120 220
```

GRID (*matrix variable name) % SIZE(number of rows*, *number of columns*)
Defines a grid consisting elements of a given matrix variable. If the given matrix variable is of type char, the elements in each row is not space-separated, otherwise they are space-separated.

For example:

```
void InputFormat() {
    GRID(G) % SIZE(2, 2);
    GRID(H) % SIZE(R, C);
}
```

With **G** = {{‘a’, ‘b’}, {‘c’, ‘d’}}, **H** = {{1, 2, 3}, {4, 5, 6}}, **R** = 2, **C** = 3, the above segments will produce:

```
ab
cd
1 2 3
4 5 6
```

6.4 Constraints

The following macros can be called inside the overridden method **BaseProblem::Constraints()** or **BaseProblem::SubtaskX()**.

CONS (*predicate*)

Defines a constraint. **predicate** is a boolean expression, whose value must be completely defined by the values of the input variables (only).

For example:

```
void Subtask1() {
    CONS(A <= B && B <= 1000);
    CONS(graphDoesNotHaveCycles());
}
```

6.5 Generator configuration

The following methods can be called inside the overridden method **BaseGenerator::Config()**.

void **setTestCasesDir** (string *testCasesDir*)

Sets the directory for the generated test case files, relative to the location of the generator program.

If not specified, the default directory is “tc”.

void **setSolutionCommand** (string *solutionCommand*)

Sets the command for executing the official solution. This will be used for generating test case output files. For each input files, this will be executed:

```
solutionCommand < [input filename] > [output filename]
```

If not specified, the default solution command is ”./solution”.

6.6 Test cases

The following macros can be called inside the overridden method **BaseGenerator::()**).

void **assignToSubtasks** (set<int> *subtaskNumbers*)

Assigns the current test test group to a set of subtasks.

For example:

```
void TestGroup1() {
    assignToSubtasks({1, 3});

    // test case definitions follow
}
```

The following macros can be called inside the overridden method **BaseGenerator::TestCases()** or **BaseGenerator::TestGroupX()**.

CASE (*comma-separated statements*)

Defines a test case.

statement should be one of:

- assignment to an input variables

- private method call that assigns values to one or more input variables

For example:

```
void TestCases() {
    CASE(N = 42, M = 100, randomArray());
    CASE(N = 1000, M = 1000, randomArray());
}
```

The following macros can be called inside the overridden method **BaseGenerator::SampleTestCases()**.

SAMPLE_CASE (*list of lines*[, *list of subtask numbers*])

Defines a sample test case. A sample test case is defined as an exact literal string, given as list of lines. **list of subtask numbers** are only valid in problems with subtasks.

For example, to define this sample test case:

```
1 2
3 4 5
```

You can do this way:

```
void SampleTestCases() {
    SAMPLE_CASE({
        "1 2",
        "3 4 5"
    });
}
```

for problems without subtasks. For problems with subtasks:

```
void SampleTestCases() {
    SAMPLE_CASE({
        "1 2",
        "3 4 5"
    }, {1, 3});
}
```

assuming that the sample test case is assigned to subtasks 1 and 3.

Multiple sample test cases can be defined inside the same method.

6.7 Random number generator

The following methods can be called on the random number generator `rnd` object inside a generator.

int **nextInt** (int *minNum*, int *maxNum*)

Return a uniformly distributed random integer (int) between minNum and maxNum, inclusive.

int **nextInt** (int *maxNumEx*)

Return a uniformly distributed random integer (int) between 0 and maxNumEx - 1, inclusive.

long long **nextLongLong** (long long *minNum*, long long *maxNum*)

Return a uniformly distributed random integer (long long) between minNum and maxNum, inclusive.

long long **nextLongLong** (long long *maxNumEx*)

Return a uniformly distributed random integer (long long) between 0 and maxNumEx - 1, inclusive.

double **nextDouble** (double *minNum*, double *maxNum*)

Return a uniformly distributed random real number (double) between minNum and maxNum, inclusive.

double **`nextDouble`**(double *maxNum*)
: Return a uniformly distributed random real number (double) between 0 and maxNum, inclusive.

void **`shuffle`**(RandomAccessIterator *first*, RandomAccessIterator *last*)
: Randomly shuffles the elements in [first, last). Use this rather than `std::random_shuffle`.

6.8 Command-line options

The following options can be specified when running the runner program. They mostly override the specified problem and generator configuration.

`--slug=slug`
: Overrides the slug specified by setSlug() in **BaseProblem::Config()**.

`--tc-dir=dir`
: Overrides the test cases directory specified by setTestCasesDir() in **BaseGenerator::Config()**.

`--solution-command=command`
: Overrides the solution command specified by setSolutionCommand() in **BaseGenerator::Config()**.

`--seed=seed`
: Sets the seed for the random number generator `rnd` inside the generator.

`--time-limit=timeLimitInSeconds`
: Overrides the time limit specified by setTimeLimit() in **BaseProblem::Config()**.

`--memory-limit=memoryLimitInMegabytes`
: Overrides the memory limit specified by setMemoryLimit() in **BaseProblem::Config()**.

`--no-time-limit`
: Unset the time limit specified by setTimeLimit() in **BaseProblem::Config()**.

`--no-memory-limit`
: Unset the memory limit specified by setMemoryLimit() in **BaseProblem::Config()**.

CHAPTER 7

Credits

tcframe is written by **Ashar Fuadi**.

It is a significant improvement of tokilib, written by the same author.

tokilib itself is essentially a wrapper for testlib, written by **Mike Mirzayanov** et al.

CHAPTER 8

License

tcframe is released under MIT license.