Rime 非公式ドキュメント

riantkb

2021 年 05 月 06 日

最初のステップ

このページに関して

このページは、プログラミングコンテストの問題準備補助ツールである Rime の **非公式** ドキュメントです。一部の記述を 公式ドキュメント から転載および改変して使用しています。

現在、 **クイックスタート** 以外のページはほとんど無が存在しています。段々と情報が書かれていく可能性があります。

第 1 章

Rime とは

Rime はプログラミングコンテストの問題セットの作成を補助するツールです。

ACM-ICPC 形式や TopCoder 形式のプログラミングコンテストを開く際、問題を出すために準備しなければならないものは何があるでしょうか。問題案は既にあるものとすると、必要なものは主に次の 4 つに大別されます。

問題文　最初から凝った文章を作る必要はもちろんありませんが、少なくとも問題の骨子と入出力形式はあらかじめ決めておかなければなりません。

模範解答プログラム　問題が解けることを証明するため、入出力データを作るため、その他あらゆる理由のために模範解答プログラムが必要です。ここで最も重要なことは、模範解答は複数用意されるべきだということです。複数の解答を複数の人間が用意して、その出力を照合することによって、解答プログラムの誤りを大幅に減らすことができます。また必須ではありませんが、想定解法と比べて計算量が大きすぎる解法や、一見正しそうに見えてエッジケースで間違う解法などの、想定「誤答」のプログラムを作成しておけば、それらを振り落とす入力データを作ることが容易になります。

入出力データ　小さな入力データは手で直接作ることができますが、大きなエッジケースやランダムケースに対応する入力データはプログラムで作ることになります。出力データは入力データを模範解答プログラムに入れて作成します。

入出力検証器　入力データのフォーマットが合っているかどうかは、模範解答プログラムとは別のプログラムであらためて検証することが望ましいでしょう。また、出力形式が浮動小数点数を含む場合など、単純な diff プログラムで間に合わない場合には、出力検証器を用意する必要もあります。

Rime は、この 4 つの項目のうち、問題文を除いた残りのすべての準備作業を補助します。たとえば、次のようなことができます。

- 手動で作成した入力データや入力ジェネレータを適切な場所に置いておけば、コマンド 1 つで入力ジェネレータを実行しデータセットを作成することができます。
- 生成された入力データセットを自動的に入力検証器に通し、フォーマット違反がないかどうかをチェックします。
- 複数の模範解答プログラムを自動的にコンパイルし、それらの出力が一致するかどうかをチェックします。

- "想定誤答プログラム" を置いておくと、そのプログラムがジャッジを通過しないことをチェックし、万が一通ってしまった場合は警告を出します。
- 各問題について、模範解答プログラムを走らせた結果を分かりやすく表示します。

第 2 章

インストール

注意: ここでは、Git と Python（および pip）がすでにインストール済みであるとして説明をします。

- もし Git がインストールされていない場合は こちら などを参考にインストールをしてください。
- もし Python がインストールされていない場合は こちら などを参考にインストールをしてください。

以下のコマンドをコマンドライン上で実行することでインストールができます（先頭の $ は入力待ちを示す記号ですので、入力する必要はありません）。

```
$ pip install git+https://github.com/icpc-jag/rime
```

インストールが正常に完了していると、rime というコマンドと rime_init というコマンドが使えるようになっているはずです。

```
$ rime --help
rime.py <command> [<options>...] [<args>...]

    Rime is a tool for programming contest organizers to automate usual, boring
    and error-prone process of problem set preparation. It supports various
    programming contest styles like ACM-ICPC, TopCoder, etc. by plugins.

    To see a brief description and available options of a command, try:

    rime.py help <command>

Commands:

build    Build a target and its dependencies.
clean    Clean intermediate files.
help     Show help.
test     Run tests in a target.
```

(次のページに続く)

(前のページからの続き)

```
Global options:

-C, --cache_tests  Cache test results.
-d, --debug        Turn on debugging.
-h, --help         Show this help.
-j, --jobs <n>     Run multiple jobs in parallel.
-k, --keep_going   Do not skip tests on failures.
-p, --precise      Do not run timing tasks concurrently.
-q, --quiet        Skip unimportant message.
```

第 3 章

クイックスタート

実際に簡単な問題の準備を行うことで、Rime の使い方に慣れていきましょう。

3.1 作業用ディレクトリの準備

まず、作問準備の作業用ディレクトリを用意しましょう。ディレクトリ名はなんでも良いです。

用意したディレクトリに移動したのち、以下を実行します。

```
$ rime_init --git
```

すると、`PROJECT` というファイルと `common/` というフォルダ（および git 管理に用いられる `.git/`, `.gitignore`）が生成されます。`PROJECT` にはいくつかの設定が書かれていますが、このチュートリアルではそれらを編集することはありません。

```
$ ls
PROJECT     common
```

3.2 新規問題の作成

それでは早速問題を作ってみましょう。今回は以下のような問題を準備することにします。

問題

整数 `A`, `B` が与えられます。 `A + B` を出力してください。

制約

`1 <= A <= 10`
`1 <= B <= 10`

まず、新規問題の作成を行います。以下のコマンドを実行してみましょう。

```
$ rime add . problem aplusb
```

上の aplusb のところが問題ディレクトリ名になります。例えば aminusb という問題ディレクトリを作りたければ、rime add . problem aminusb とすれば良いです。詳しくはこちら [TODO] をご覧ください。

> **注意:** 上のコマンドを実行すると、コンソールが謎の文字列に覆われて再起不能に見えるようになるかもしれません。これは問題の設定ファイルをその場で編集できるようになっており、それに用いられる既定のエディタが Vim に設定されていることに起因するものです。もし Vim の使い方に明るくないならば、慌てずに :q と打ってエンターキーを押しましょう。

すると、aplusb/ というディレクトリが作られます。このディレクトリに移動してみましょう。中には、PROBLEM というファイル 1 つだけが入っています。

```
$ ls
PROJECT    aplusb     common
$ cd aplusb/
$ ls
PROBLEM
```

PROBLEM ファイルの中身は以下のようになっています。このファイルも、6 行目の time_limit（Rime におけるその問題の実行時間制限）の値を適宜変更する以外は基本的に編集する必要はありません。

リスト 1　PROBLEM

```
# -*- coding: utf-8; mode: python -*-

pid='X'

problem(
  time_limit=1.0,
  id=pid,
  title=pid + ": Your Problem Name",
  #wiki_name="Your pukiwiki page name", # for wikify plugin
  #assignees=['Assignees', 'for', 'this', 'problem'], # for wikify plugin
  #need_custom_judge=True, # for wikify plugin
  #reference_solution='???',
)

atcoder_config(
  task_id=None # None means a spare
)
```

ちなみに: 12 行目の reference_solution のコメントアウトを解除して適切に設定することで、想定解出力にどの解答プログラムを利用するかを選択することもできます。複数の解答プログラムで速度に差がある場合や、正答となる出力が複数存在する場合などに役立つかもしれません。

詳しくは こちら [TODO] を参照してください。

3.3 解答プログラムの作成

次に、解答プログラムを作成してみます。先ほど作成した aplusb/ ディレクトリ内で、以下のコマンドを実行してみましょう。

```
$ rime add . solution cpp_correct
```

上の cpp_correct のところが解答プログラムのディレクトリ名になります。ここの名前はなんでも良いです。

すると、(エディタが起動したのち、) cpp_correct/ というディレクトリが作られます。このディレクトリに移動してみましょう。中には、SOLUTION というファイル 1 つだけが入っています。

```
$ ls
PROBLEM     cpp_correct
$ cd cpp_correct/
```

(次のページに続く)

(前のページからの続き)

```
$ ls
SOLUTION
```

SOLUTION ファイルの中身は以下のようになっています。

リスト 2　SOLUTION

```
# -*- coding: utf-8; mode: python -*-

## Solution
#c_solution(src='main.c') # -lm -O2 as default
#cxx_solution(src='main.cc', flags=[]) # -std=c++11 -O2 as default
#kotlin_solution(src='main.kt') # kotlin
#java_solution(src='Main.java', encoding='UTF-8', mainclass='Main')
#java_solution(src='Main.java', encoding='UTF-8', mainclass='Main',
#              challenge_cases=[])
#java_solution(src='Main.java', encoding='UTF-8', mainclass='Main',
#              challenge_cases=['10_corner*.in'])
#rust_solution(src='main.rs') # Rust (rustc)
#go_solution(src='main.go') # Go
#script_solution(src='main.sh') # shebang line is required
#script_solution(src='main.pl') # shebang line is required
#script_solution(src='main.py') # shebang line is required
#script_solution(src='main.rb') # shebang line is required
#js_solution(src='main.js') # javascript (nodejs)
#hs_solution(src='main.hs') # haskell (stack + ghc)
#cs_solution(src='main.cs') # C# (mono)

## Score
#expected_score(100)
```

この中で、解答プログラムの言語に対応した行のコメントアウトを解除し、必要に応じてソースファイル名を変更します。今回は C++ の解答プログラムを追加するため、5 行目のコメントアウトを解除します。ファイル名については、今回は `ans.cpp` としてみます。

リスト 3　SOLUTION（一部抜粋）

```
cxx_solution(src='ans.cpp', flags=[]) # -std=c++11 -O2 as default
```

ちなみに: 実は、ここでコメントアウトを解除しなくとも、Rime はディレクトリ内のファイルの拡張子を参照することで解答プログラムの言語をよしなに解釈してくれます。ただ、想定誤解法の追加時にはここの設定が必須なので慣れておけると良いでしょう。

それでは、次は実際の解答プログラムを追加します。ここでは、`cpp_correct/` ディレクトリ内に自分でファイルを作成してプログラムを書きます。この問題では、例えば以下のようなプログラムになるでしょう。

リスト 4　ans.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b << '\n';
    return 0;
}
```

3.4 テスト用プログラムの作成

次に、テスト用プログラムを作成します。ここで言うテスト用プログラムとは、解答プログラムが正しく問題を解決するプログラムであるかどうかを判断するためのプログラムの総称であり、Rime においてユーザーが用意する必要のあるプログラムは主に以下の 2 つです。

入力生成器 (generator)　解答プログラムに与える入力を生成するプログラム

入力検証器 (validator)　解答プログラムに与える入力が問題の制約を正しく満たしているかを検証するプログラム

ちなみに: 想定される出力が複数ある場合や出力された実数の誤差を許容する場合などに、加えて **出力検証器 (judge)** が必要になることもあります。

それでは、テスト用プログラムを作成していきます。 `aplusb/` ディレクトリ内で、以下のコマンドを実行してみましょう。

```
$ rime add . testset tests
```

上の `tests` のところがテスト用プログラムのディレクトリ名になります。ここの名前はなんでも良いですが、慣例的に `tests` という名称が用いられることが多いです。

すると、(エディタが起動したのち、) `tests/` というディレクトリが作られます。このディレクトリに移動してみましょう。中には、`TESTSET` というファイル 1 つだけが入っています。

```
$ ls
PROBLEM     cpp_correct tests
$ cd tests/
$ ls
TESTSET
```

TESTSET ファイルの中身は以下のようになっています。

リスト 5　TESTSET

```
# -*- coding: utf-8; mode: python -*-

## Input generators.
#c_generator(src='generator.c')
#cxx_generator(src='generator.cc', dependency=['testlib.h'])
#java_generator(src='Generator.java', encoding='UTF-8', mainclass='Generator')
#rust_generator(src='generator.rs')
#go_generator(src='generator.go')
#script_generator(src='generator.pl')

## Input validators.
#c_validator(src='validator.c')
#cxx_validator(src='validator.cc', dependency=['testlib.h'])
#java_validator(src='Validator.java', encoding='UTF-8',
#               mainclass='tmp/validator/Validator')
#rust_validator(src='validator.rs')
#go_validator(src='validator.go')
#script_validator(src='validator.pl')

## Output judges.
#c_judge(src='judge.c')
#cxx_judge(src='judge.cc', dependency=['testlib.h'],
#          variant=testlib_judge_runner)
#java_judge(src='Judge.java', encoding='UTF-8', mainclass='Judge')
#rust_judge(src='judge.rs')
#go_judge(src='judge.go')
#script_judge(src='judge.py')

## Reactives.
#c_reactive(src='reactive.c')
#cxx_reactive(src='reactive.cc', dependency=['testlib.h', 'reactive.hpp'],
#             variant=kupc_reactive_runner)
#java_reactive(src='Reactive.java', encoding='UTF-8', mainclass='Judge')
#rust_reactive(src='reactive.rs')
#go_reactive(src='reactive.go')
#script_reactive(src='reactive.py')

## Extra Testsets.
# icpc type
#icpc_merger(input_terminator='0 0\n')
# icpc wf ~2011
#icpc_merger(input_terminator='0 0\n',
#            output_replace=casenum_replace('Case 1', 'Case {0}'))
#gcj_merger(output_replace=casenum_replace('Case 1', 'Case {0}'))
id='X'
#merged_testset(name=id + '_Merged', input_pattern='*.in')
```

(次のページに続く)

(前のページからの続き)

```
#subtask_testset(name='All', score=100, input_patterns=['*'])
# precisely scored by judge program like Jiyukenkyu (KUPC 2013)
#scoring_judge()
```

この中で、テスト用プログラムの言語に対応した行のコメントアウトを解除し、必要に応じてソースファイル名を変更します。今回は C++ の generator, validator を追加するため、それぞれ該当する行のコメントアウトを解除します。

リスト 6　TESTSET（一部抜粋）

```
cxx_generator(src='generator.cc', dependency=['testlib.h'])
```

```
cxx_validator(src='validator.cc', dependency=['testlib.h'])
```

さて、それでは入力生成器と入力検証器を作成していきます。これらは、Rime では testlib というライブラリを用いて書かれることが多いです。

`tests/` ディレクトリ内に、`generator.cc` と `validator.cc` を追加します。入力生成器、入力検証器の詳しい仕様については こちら [TODO] をご覧ください。

リスト 7　generator.cc

```
#include <iostream>
#include "testlib.h"
using namespace std;

const int MIN_A = 1;
const int MAX_A = 10;
const int MIN_B = 1;
const int MAX_B = 10;

int main(int argc, char** argv) {
    registerGen(argc, argv, 1);
    for (int t = 0; t < 10; t++) {
        ofstream of(format("02_random_%02d.in", t + 1).c_str());
        int a = rnd.next(MIN_A, MAX_A);
        int b = rnd.next(MIN_B, MAX_B);
        of << a << ' ' << b << '\n';
        of.close();
    }
    return 0;
}
```

リスト 8　validator.cc

```cpp
#include <iostream>
#include "testlib.h"
using namespace std;

const int MIN_A = 1;
const int MAX_A = 10;
const int MIN_B = 1;
const int MAX_B = 10;

int main(int argc, char** argv) {
    registerValidation(argc, argv);
    inf.readInt(MIN_A, MAX_A, "A");
    inf.readSpace();
    inf.readInt(MIN_B, MAX_B, "B");
    inf.readEoln();
    inf.readEof();
    return 0;
}
```

上の入力生成器はランダムな入力を生成しますが、それ以外にサンプル入力やコーナーケースなど手で作ったケースを入れたくなるかもしれません。そういう場合は、 `tests/` ディレクトリ以下に `.in` という拡張子で入力ファイルを置いておくことで入力に含めることができます。

3.5 テストの実行

ようやく準備が整ったので、テストを実行します。 `aplusb/` ディレクトリに戻り、以下のコマンドを実行してみましょう。

```
$ rime test
[ COMPILE  ] aplusb/tests: generator.cc
[ COMPILE  ] aplusb/tests: validator.cc
[ GENERATE ] aplusb/tests: generator.cc
[ VALIDATE ] aplusb/tests: OK
[ COMPILE  ] aplusb/cpp_correct
[  REFRUN  ] aplusb/cpp_correct
[   TEST   ] aplusb/cpp_correct: max 0.00s, acc 0.03s

Build Summary:
aplusb ... in: 40B, diff: 25B, md5: -
  cpp_correct CXX 9 lines, 130B

Test Summary:
```

(次のページに続く)

(前のページからの続き)

```
aplusb ... 1 solutions, 10 tests
  cpp_correct  OK  max 0.00s, acc 0.03s

Error Summary:
Total 0 errors, 0 warnings
```

注意: 謎のコンパイルエラーでテストができない場合

ひょっとしてあなたはいま Mac を使っていて、かつ `bits/stdc++.h` をインクルードしていませんか？ Rime では C++ のコンパイル時に環境変数 `CXX` を参照し、定義されていない場合は `g++` を使用します。Mac では `g++` と打つと clang が動くので `bits/stdc++.h` が無いと言われてしまいます。解決策としては `bits/stdc++.h` を使わないか、もしくは以下のように環境変数 `CXX` を指定してあげれば良いです（`g++-10` のところは、必要に応じてインストールされている GCC のコマンド名に置き換えてください）。

```
$ CXX=g++-10 rime test
```

無事にテストをすることができました。

第 4 章

次のステップ

4.1 誤答が落ちることをテストする

クイックスタート では「正しいプログラムが正しく受理されること」をテストしましたが、これで本当に十分でしょうか。実際には、「正しくないプログラムが正しく落ちること」も同じだけ重要なので、このことについてもテストを行うべきです。

この項では、 **クイックスタート** の **新規問題の作成** で作成した A + B の問題を用いて誤答のテストの例を示します。

まず、 **クイックスタート** の **解答プログラムの作成** と同様に解答プログラムを作成します。 `aplusb/` ディレクトリ内で、以下のコマンドを実行しましょう。

```
$ rime add . solution cpp_wa
```

次に、生成された `SOLUTION` ファイルを編集します。解答プログラムの言語に対応した行のコメントアウトを解除し、必要に応じてソースファイル名を変更するところまでは **クイックスタート** と同様です。今回は C++ の解答プログラムを追加するため、5 行目のコメントアウトを解除します。ファイル名については、 `ans.cpp` としてみます。

ここで、 `cxx_solution` の引数に `challenge_cases=[]` を追加します。このようにすることで、「このプログラムは落ちなければならない」ということを指定することができます。

リスト 1　SOLUTION（一部抜粋）

```
5 cxx_solution(src='ans.cpp', flags=[], challenge_cases=[]) # -std=c++11 -O2 as default
```

ちなみに: `challenge_cases` は、正確には `challenge_cases=['10_corner_01.in']` などというように入力ファイルをいくつか指定し、そのうち 1 つ以上で正答を返さないことをテスト通過の条件とするものです。

入力ファイルを 1 つも指定しない場合、指定された入力ファイルが入力ファイル全体となるため、ほとんどの場合は 1 つも指定しない使い方で事足りるでしょう。

それでは、実際に ans.cpp を変更して正しい答えを返さないプログラムを用意してみましょう。今回は、答えが偶数のときに 1 大きい答えを出力するように変更してみます。

リスト 2　ans.cpp

```cpp
#include<iostream>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    int ans = a + b;
    if (ans % 2 == 0) {
        ans++;
    }
    cout << ans << '\n';
    return 0;
}
```

それでは aplusb/ ディレクトリに戻り、テストを実行してみましょう。

```
$ rime test
[ COMPILE  ] aplusb/tests: generator.cc
[ COMPILE  ] aplusb/tests: validator.cc
[ GENERATE ] aplusb/tests: generator.cc
[ VALIDATE ] aplusb/tests: OK
[ COMPILE  ] aplusb/cpp_correct
[  REFRUN  ] aplusb/cpp_correct
[   TEST   ] aplusb/cpp_correct: max 0.01s, acc 0.05s
[ COMPILE  ] aplusb/cpp_wa
[   TEST   ] aplusb/cpp_wa: 02_random_04.in: Wrong Answer

Build Summary:
aplusb ... in: 40B, diff: 25B, md5: -
  cpp_correct CXX  9 lines, 130B
  cpp_wa      CXX 13 lines, 194B

Test Summary:
aplusb ... 2 solutions, 10 tests
  cpp_correct  OK  max 0.01s, acc 0.05s
  cpp_wa       OK  02_random_04.in: Wrong Answer

Error Summary:
Total 0 errors, 0 warnings
```

Test Summary の cpp_wa の部分を見ると、 02_random_04.in: Wrong Answer と書いてあり、02_random_04.in というケースで間違った答えを出力したのでテストは成功した、ということがわかります。

次に、誤答プログラムを少し変更してみましょう。今度は答えが 20 のときに RE となる（終了ステータスが 0 でない）ように変更してみます。

リスト 3　ans.cpp

```cpp
#include<iostream>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    int ans = a + b;
    if (ans == 20) {
        return 1;
    }
    cout << ans << '\n';
    return 0;
}
```

テストを実行してみます。

```
$ rime test
[   TEST   ] aplusb/cpp_correct: max 0.01s, acc 0.04s
ERROR: aplusb/cpp_wa: Unexpectedly accepted all test cases
[   TEST   ] aplusb/cpp_wa: Unexpectedly accepted all test cases

Build Summary:
aplusb ... in: 40B, diff: 25B, md5: -
  cpp_correct CXX  9 lines, 130B
  cpp_wa      CXX 13 lines, 194B

Test Summary:
aplusb ... 2 solutions, 10 tests
  cpp_correct  OK  max 0.01s, acc 0.04s
  cpp_wa      FAIL Unexpectedly accepted all test cases

Error Summary:
ERROR: aplusb/cpp_wa: Unexpectedly accepted all test cases
Total 1 errors, 0 warnings
```

生成したランダムケースには A = B = 10 のケースは入っていなかったらしく、全てのテストケースで正答を返してしまいました。

それでは、このプログラムを落とせるケースを追加してみましょう。そのような入力を生成する入力生成器を追加しても良いのですが、今回は手で作ったケースを追加することで実現してみます。

aplusb/tests/ ディレクトリの中に 05_corner_01.in というファイルを追加し、A = B = 10 のケースを用意します。

リスト 4　05_corner_01.in

```
10 10
```

この状態で、再びテストを行ってみます。

```
$ rime test
[ COMPILE  ] aplusb/tests: generator.cc
[ COMPILE  ] aplusb/tests: validator.cc
[ GENERATE ] aplusb/tests: generator.cc
[ VALIDATE ] aplusb/tests: OK
[  REFRUN  ] aplusb/cpp_correct
[   TEST   ] aplusb/cpp_correct: max 0.01s, acc 0.06s
[   TEST   ] aplusb/cpp_wa: 05_corner_01.in: Runtime Error

Build Summary:
aplusb ... in: 46B, diff: 28B, md5: -
  cpp_correct CXX  9 lines, 130B
  cpp_wa      CXX 13 lines, 194B

Test Summary:
aplusb ... 2 solutions, 11 tests
  cpp_correct  OK  max 0.01s, acc 0.06s
  cpp_wa       OK  05_corner_01.in: Runtime Error

Error Summary:
Total 0 errors, 0 warnings
```

きちんと RE が出て、テストが成功しました。

ちなみに: challenge_cases で誤答であるということを指定する以外に、WA, RE, TLE などのうちどの判定になってほしいか、という期待する判定を指定することもできます。

詳しくは こちら (TODO) を参照してください。

4.2 制約違反の入力が検出できることを確認する

todo

wip

4.3 入出力ファイルを出力する

todo

wip

第 5 章

ジャッジが特殊な問題を作りたい

5.1 スペシャルジャッジ

---

todo

wip

5.2 リアクティブジャッジ

---

todo

wip

5.3 スコアリングジャッジ

---

todo

wip

第 6 章

コマンドライン

6.1 rime

---

todo

wip

```
rime COMMAND [OPTIONS] [ARGS]
```

6.1.1 rime help

```
rime help [COMMAND]
```

6.1.2 rime test

```
rime test [TARGET]
```

6.1.3 rime clean

```
rime clean [TARGET]
```

6.2 rime_init

todo

wip

```
rime_init [--git | --mercurial]
```

--git

第 7 章

設定ファイル

7.1 PROJECT

---

todo

wip

7.2 PROBLEM

---

todo

wip

7.3 SOLUTION

---

todo

wip

7.4 TESTSET

todo

wip

索引