Мы будем говорить во основном о стресс-тестировании в олимпиадном программировании, где нужно все делать максимально быстро и эффективно. Стресс-тестирование — это один из способов находить ошибку в программе. Его преимущества заключаются в том, что его легко написать и использовать и с его помощью можно найти контртест, на котором наше решение не работает.

1 Типы задач

Буквально два слова об этом. Это важно, так для каждого вида стресс-тестирование будет немного отличаться. В олимпиадном программировании есть преимущественно три типа задач.

1.1 Задачи с единственным правильным ответом

В таких задачах ответ всегда совпадает (или находится в пределах погрешности) с ответом жюри.

1.2 Задачи с множеством правильных ответов

В таких задачах как правило просят найти один из оптимальных ответов, которых может быть несколько.

1.3 Интерактивные задачи

В таких задач наша программа работает с интерактором, то есть ее выполнение зависит от "скрытых"данных и вывода программы жюри.

2 Программа

Для начала давайте поймем, как устроены программы в олимпиадном программировании. Весь код находится в одном файле и состоит из нескольких важных для нас частей. Необязательно четко разделять их в программе на отдельные блоки. Это можно делать при необходимости.

2.1 Глобальные переменные

Глобальные переменные инициализируются в начале выполнения программы и существуют до самого конца. Об этом следует не забывать в задачах с мультитестом (когда во время одного выполнения программы проверяется несколько независимых наборов данных).

2.2 Ввод

Просто часть программы, в которой мы вводим данные теста. Его удобно писать в отдельной функции, если ввод имеет сложный формат.

2.3 Решение

Существенная часть программы, в которой мы выполняем алгоритмическую часть решения задачи.

2.4 Вывод

Часть программы, в которой мы оформляем результат работы нашей программы для сравнения с правильным ответом. Как правило, эта часть достаточно простая, но в ней также может быть, например, восстановление ответа в ДП.

3 Ошибки

Теперь давайте поговорим о том, какие могут быть ошибки в программе и как их можно исправить.

3.1 Неправильно прочитанное условие

Сюда мы отнесем ошибки, которые связаны с неправильным пониманием условия. В данном случае ошибки зачастую являются осознанными, поэтому если вы слишком долго ищите ошибку, возможно, стоит перечитать условие заново.

3.1.1 Легенда

Иногда легенда задачи может путать, поэтому при перечитывании лучше сакцентировать свое внимание на формальной части условия. Как правило, она находиться в конце текста.

3.1.2 Ввод

Необходимо проверять, что все константы в коде соответствуют ограничениям, а также некоторые части ввода, в которых легко что-то перепутать (например, длину и ширину, перестановку и порядок индексов).

3.1.3 Вывод

Необходимо проверять, что вы выводите действительно то, что от вас требуется. Как правило, формат вывода достаточно простой и ошибки связанные с ним встречаются редко (но это не значит, что они менее опасные!).

3.2 Различные компиляторы

Различные версии компиляторов на вашем компьютере и в тестирующей системе могут привести к неожиданным эффектам. Например, в VS можно обращаться к методу нулевого указателя, а в GNU G++ нельзя, в Linux и Windows rand() имеет разный диапазон. Чтобы избежать нежелательных ошибок такого рода, нужно либо разбираться в тонкостях GNU G++ (он есть на любой олимпиаде), либо выбирать в тестирующей системе компилятор, аналогичный стоящему на компьютере.

3.3 Wrong Answer (WA)

Мы будем считать, что в программе есть ошибка, если она получает вердикт в тестирующей системе, отличный от правильного. Исходя из того, какой вердикт был получен, можно частично понять, где находится ошибка. Одним из самых частых результатов проверки теста является неправильный ответ (WA). В данном случае, к сожалению, ошибка может быть где угодно, поэтому часто приходиться искать тест, на котором наша программа не работает, и дебажить его.

3.4 Runtime Error (RE)

Кроме WA также бывает и RE — ошибка выполнения. Это когда программа пытается сделать то, чего не может, и "падает"из-за этого. Вот список причин, из-за которых чаще всего случается RE:

1. Выход за границы массива
2. Обращение к нулевому указателю
3. Некорректные операции с числами
4. Переполнение стека рекурсии

Такие ошибки часто можно найти "глазами", но если не получается, можно так же найти тест, на котором решение не работает, и отдебажить его. Для ускорения поиска ошибки можно использовать отладочные макроопределения (по сути, встроены в дебаггер VS) или санитайзеры.

3.5 Time Limit (TL)

Еще один возможный вердикт — это TL. Такой статус посылки бывает в случаях, когда программа работает слишком долго. Если у вас TL, стоит проверить асимптотическую сложность алгоритма, а также нет ли зацикленности или бесконечной рекурсии.

3.6 Memory Limit (ML)

ML — это превышение ограничения по памяти. В целом, проблемы те же, что и в TL.

4 Когда стоит начать писать стресс-тест

Стресс-тестирование — это способ точно найти контртест к своему решению. С помощью этого теста легко локализовать и устранить ошибку в коде. Но, чтобы написать стресс-тест, потребуется некоторое время — от 10 до 30 минут, в зависимости от сложности задачи. Поэтому самым лучшим решением будет поставить некоторый "дедлайн", например, 7 минут, в течение которого вы можете искать ошибку любыми удобными вам способами (перечитывать код, перепроверять правильность решения, пытаться найти тест и др.). Если за отведенное время так ничего и не вышло, стоит начать писать стресс-тест. Его преимущества над другими методами также в том,

что его можно использовать для всех решений задачи и что он поможет найти сразу все ошибки, а не только какую-то одну.

5 Стресс-тестирование внутри программы

Самый простой способ написать стресс-тест — это написать его прямо в коде в виде функции, которая будет бесконечно генерировать тесты и проверять правильность работы программы. Недостатки данного решения в том, что количество кода сильно возрастет, и в нем будет легче запутаться. Я рекомендую выделять каждую часть кода (глобальные переменные, ввод, решение, вывод) в отдельные блоки или создавать копию решения без ввода-вывода (но тогда нужно будет не забыть исправить ошибки в оригинальном решении).
Теперь давайте разберемся, как лучше всего писать стресс-тест для каждого типа задачи.

5.1 Задачи с единственным правильным ответом

Здесь нам обязательно понадобиться решение, которое выдает правильный ответ. Оно необязательно должно быть оптимальным, то есть проходить по TL или ML, поэтому его будет легко написать (как правило, работает просто перебор). Теперь, все, что нам нужно, это уметь генерировать много тестов и сравнивать, что ответы одинаковые. Как только они не совпадут (или программа неожиданно упала или зациклилась), это будет значить, что мы нашли контртест к своему решению. Дальше уже дело техники.
Итак, наша программа делится на несколько частей:

1. Наше решение, в котором мы ищем ошибку
2. Правильное решение, в котором мы уверены
3. Генератор
4. Стресс-тест

Сам стресс-тест можно оформить примерно следующим образом:

```
void stresstest() {
  int test_n = 1;
  while (true) {
    cout << "TEST #" << test_n++ << '\n';
    gen_test();
    if (solve() != stupid_solve()) {
      cout << "FAIL\n";
      print_test();
      cout << "MY ANS: " << solve() << '\n';'
      cout << "R8 ANS: " << stupid_solve() << '\n';
    } else {
      cout << "SUCCES\n";
    }
  }
}
```

5.2 Задачи с множеством правильных ответов

Главное отличие этого типа задач от предыдущего — это возможное неполное совпадение ответов. То есть кроме проверки, что ответ оптимальный, нужно проверять и его корректность. Поэтому у нас добавиться еще одна функция в стресс-тест — чекер. Если нужно проверять только корректность ответа (а в его оптимальности мы уверены), то правильное решение можно не писать — это сэкономит много времени.
Как проверить, что проблема именно в корректности ответа? Можно, например, написать чекер заранее и вызывать его через встроенную функцию "assert()". Тогда, если вдруг ответ некорректен, вердикт сменится с WA на RE.

5.3 Интерактивные задачи

Я рекомендую писать для интерактивных задач отдельную функцию "ask()", которая будет взаимодействовать с интерактором. Во-первых, так будет удобно заниматься отладкой кода, ведь для этого нужно будет заменить код лишь в одном месте. А во-вторых, будет легко написать стресс-тест. В данном типе задач нам нужен будет только чекер, который обычно очень просто выглядит. Функция запроса может выглядеть примерно следующим образом:

```
const int MAX_QUERY = 100;
int ja[N];  // jury's array
int ask(int i) {
  static int cnt = 0;
  assert(cnt++ < MAX_QUERY);
  cout << "? " << i + 1 << endl;
  int x;
  if (_DEBUG) {
    cout << ja[i] << endl;
    x = ja[i];
  } else {
    cin >> x;
  }
  return x;
}
```

6 Стресс-тестирование через subprocess

Удобство данного способа заключается в том, что нам не придется никак изменять исходный код решения, а только написать новый. Особенно такое решение может быть удобно для интерактивных задач. Минусы: нужно знать Python и такой способ скорее всего займет немного больше времени. В питоне есть встроенная библиотека "subprocess с помощью которой можно вызвать

```
solve = subprocess.Popen([solve_path],
            stdin=subprocess.PIPE, stdout=subprocess.PIPE)
# we can write something for the solve's stdin
solve.stdin.write("5\n".encode())
solve.stdin.flush()
# we can get output of the solve's stdout
ans = solve.stdout.readline().decode().strip()
```

Пример возможной реализации интерактора можно найти здесь.

7 Стресс-тестирование в Linux

В Linux'е очень удобно использовать внешний стресс-тест и для этого не нужно много чего знать. Обычный стресс-тест будет выглядеть примерно следующим образом:

```
# compile.sh
g++ -O2 -Wall -Wextra -Wconversion $1.cpp -o $1
```

```
# stresstest.sh
./compile.sh gen
./compile.sh solve
./compile.sh stupid

for t in $(seq 1 1000); do
  echo "Test $t"
  ./gen $t > test || { echo "gen failed"; exit; }
  ./solve < test > out || { echo "solve failed"; exit; }
  ./stupid < test > ok || { echo "stupid failed"; exit; }
  diff ok out || exit
done
```

Программам нужно давать разрешение на компиляцию:

```
$ chmod +x
```

Чтобы вывести содержимое файла, нужно написать следующую команду:

```
$ cat test
```

8 Стресс-тестирование с помощью .cmd

Спасибо Даниле Белоусу за то, что рассказал про этот способ.
В Windows можно писать стресс-тесты подобным образом, создав файл с расширением .cmd:

```
@echo off
for /L %%i in (1, 1, 100000) do (
  A_gen.exe %%i >input.txt
  A_to_check.exe <input.txt >output.txt
  A_ok.exe <input.txt >outputok.txt
  echo %%i
  FC output.txt outputok.txt
  if errorlevel 1 (
  exit
  )
)
```

9 Лайфхаки с генератором

1. Тесты небольшого размера хорошо ищут крайние тесты, а также их удобнее дебажить.

2. Необязательно делать абсолютно все переменные рандомными, некоторые можно оставлять константами (например, говорить, что размер массива всегда равен 7).

3. Лучше запускать стресс-тест всегда с одинаковым начальным сидом. Если стресс-тест проходит первые много тестов, бессмысленно менять сид, лучше попробовать поменять переменные в генераторе.

4. Необязательно писать сложные равновероятные генераторы тестов.

5. Нужно иметь в виду, что rand() в Windows имеет маленький диапазон чисел (от 0 до $2^{15} - 1$). Если нужен диапазон чисел больше можно написать следующую функцию:

```
int rnd() { return rand() << 15 ^ rand(); }
```

   Или просто использовать другой генератор случайных чисел mt19937:

```
#include <random>

mt19937 rnd(7);
```

10 Заключение

Мы рассмотрели несколько способов написания стресс-теста — проверки своего решения генерацией большого числа тестов. Метод поиска ошибки с его помощью может занять некоторое время, поэтому логично сначала перечитать в коде сомнительные моменты, и только после приступать к стресс-тестированию. Главные умения, которые пригодятся при таком подходе — это написание хорошего генератора и быстрый дебаг.