What to do if your solution doesn't work?

You've coded everything perfectly, tested your solution on the tests included in the statements and happily submit it to the system, but your solution fails on test #whatever and the system doesn't even show what test is failing!

The verdict is usually one the following

1. Wrong answer
2. Time/memory limit exceeded
3. Failed (runtime error)

Although the strategies may differ depending on the verdict, generally you need to follow the same steps.

- Check if you didn't forget some corner case. For example if $n = 1$ or n is the maximum possible number satisfying the constraints. Check if your program behaves correctly given the input that hits constraints in all possible senses.
- Design some general test(s) that you know the correct answer for. **Don't** look at your code's answer for this tests before you figure out what the answer should be with pen and paper. Otherwise it is easy to convince yourself that it is correct and fail to see some silly error.
- If you got time limit exceeded error measure how long your program works for the larger inputs. The following functions help measure the CPU time since the start of the program:

  C++ `(double)clock() / CLOCKS_PER_SEC` with `ctime` included.

  python `time.clock()` returns floating-point value in seconds.

  Java `System.nanoTime()` returns long value in nanoseconds.

  Measure time for small tests, medium tests and large tests. You may encounter one of the possible outcomes:

  - Your program works for small and medium tests in time, but for larger tests it is more than 10 times slower than needed (or just hangs for the large tests). In that case you probably have a complexity issues. You may want to:
    * Measure the time parts of the program take separately (for example, how much time reading the input/printing the output take)
    * Compute actual number of operations your algorithm and its parts do and see if it is as expected.

    - Check if you pass references to your functions (only applies to c++, in java and python it is always references)
  - Your program hangs on a small or a medium test. You want to check for an infinite loop/recursion. Add assertions (`assert` in c++, python and java) on preconditions and postconditions of your loops and functions and see if they fail. Use debug output/debugger to see what code path leads to the hang.

- If you got a runtime error (usually the message is Unknown signal ...) then it is good news, that is the most informative verdict, that means your program crushes due to one of the following factors:
  - You access a location in memory that doesn't belong to your program. In C++ it can take two forms: you are trying to access a non-existing element of an array, you are trying to evaluate a null-pointer or a pointer that points to a location that doesn't belong to your program.
  - You make an arithmetic error: division by zero, overflow of a floating-point number.
  - *(C++ specific)* You exceed the stack size, that may either be caused by infinite recursion or by creating a large object (such as array) inside a function.

  Generate different tests (see page 2) and run your program against them until it crushes.

- The 'wrong answer' verdict is probably the most challenging, there are many things that can lead to this verdict. In order to find a failing test you can do one of the following things:
  - Find an alternative solution that may not be correct in terms of efficiency (and in some cases doesn't even work for some types of tests) but may be used to check if your main solution's answer is correct.
  - Make your program crush if something is inconsistent. That means adding assertions for postconditions and preconditions of your functions and loops (see page 5 for details).

How to generate tests?

The simplest way to generate test is to write a program that prints a test to a text file. For example

Algorithm 1: Generating a test for Maximum Pairwise product

```
import sys
n = int(sys.argv[1])
print(n)
print(' '.join([str(i*2) for i in range(n)]))
```

The most cryptic thing here is probably `sys.argv[1]` this is a getter for the first command line argument. In order to run your program with a command line argument you should open a terminal. For windows press Win+R, type "cmd" and press Enter, for linux press ctrl+alt+T, for MacOS press Command+spacebar to launch Spotlight and type "Terminal" then double-click the search result. Then navigate to a directory (using "cd" command) you've saved the above script in, type `python3 script.py 10` (where script.py is the name of your script) and press Enter.

Now, how use this to run your program? Copy the executable file or a python script to the same directory as your generating script and run the following commands:

```
python script.py 17 > input.txt
./your_program_name < input.txt
```

If your program is written in python instead of `./your_program_name` use `python3 your_program_name`. The

```
> input.txt
```

command redirects the output of the generating script to a text file and

```
< input.txt
```

redirect the input of your program to the same file.

So, now you can automatically generate tests and run your program against them, but the following questions remain unanswered:

- How to randomize your tests?
- How to generate a lot of them?
- How to check the answers?

Generating random tests and running your program on them

We are going to use the following technique consisting of three parts:

1. Test generator that accepts a seed as a command line parameter: Algorithm 2;
2. An alternative solution;
3. A script that repeatedly generates a test with the generator from (1), then runs both the main solution and the model solution on the generated test and checks if the answers coincide: Algorithm 3.

Algorithm 2: Generator that accepts a seed from the command line

```
import random
import sys

n = int(sys.argv[1])
myseed = int(sys.argv[2])
random.seed(myseed)

print(n)
# 1000 could also be moved to parameters instead of making it
# a hard constant in the code
print(' '.join([str(random.randint(1,1000)) for i in range(n)]))
```

For the main script that connects all the parts together we are going to use very simple and crude method `os.system(s)` that simply runs the string `s` in the command line:

Algorithm 3: Main script

```
import random
import sys
import os

# accept the number of tests as a command line parameter
tests = int(sys.argv[1])
# accept the parameter for the tests as a command line parameter
n = int(sys.argv[2])


for i in range(tests):
  print("Test #" + str(i))
  # run the generator gen.py with parameter n and the seed i
  os.system("python3 gen.py " + str(n) + " " + str(i) + " > input.txt")
  # run the model solution model.py
  # Notice that it is not necessary that solution is implemented in
  # python, you can as well run ./model <input.txt >model.txt for a C++
  # solution.
  os.system("python3 model.py <input.txt >model.txt")
  # run the main solution
  os.system("python3 main.py <input.txt >main.txt")

  # read the output of the model solution:
  with open('model.txt') as f: model = f.read()
  print("Model: ", model)
  # read the output of the main solution:
  with open('main.txt') as f: main = f.read()
  print("Main: ", main)
  if model != main:
    break
```

How to add assertions?

In Java, Python and C++ `assert expr;`, `assert expr`, and `assert(expr);` respectively is going to produce a runtime error if the Boolean expression `expr` is false. One possible usage of assertions is verifying that the answer of your program is consistent. Another use case which is more common is to verify that intermediate steps of your program are consistent.