Introduction and learning outcomes

Please note that this reading accompanies the three video lectures about Stress Testing.

In this reading we will introduce a very powerful technique called stress testing which will help you both in the situation when you are stuck with a wrong program and when you want to get the solution to the programming assignment accepted from the first attempt.

In this reading you will...

1. Review the testing techniques to apply before stress testing.
2. Learn the general scheme of stress testing a solution of an algorithmic problem.
3. Review a sample implementation of stress testing for the maximum pairwise product problem.
4. Use stress test to automatically find a test on which your solution gives wrong answer.
5. Find a small and simple test on which the solution breaks.
6. Debug the solution on the simple test case.
7. Finally fix the solution of the maximum pairwise product problem, submit it and pass all the tests.

Testing techniques

In the previous section, we've already done some testing and fixed some mistakes. In general, you should test your problem on the following groups of tests before submitting:

1. A few small manual tests.
2. A test for each possible type of answer (smallest answer, biggest answer, answer doesn't exist, etc.)
3. Test for time/memory limit: generate a test with the largest possible size of input ("max test"), run your program, measure time and memory consumption.
4. Tests for corner cases:

- Smallest possible "n": the length of the input sequence or string, the number of queries, etc.
- Equal numbers, equal letters in the string, more generally, equal objects in the input. Any two objects that are not restricted to be different in the problem statement can be equal.
- Largest numbers in the input allowed by the problem statement - for example, to test for integer overflow.

- Degenerate cases like empty set, three points on one line, a tree which consists of just one chain of nodes.

If after all of that you're sure that all answers are correct, but your solution is not accepted in the testing system, and you don't know what is the test case where your solution fails, there's the last resort called **stress testing** - a very efficient technique that will find you a test case for which your solution fails probably in 95% of cases not covered by the previously mentioned test types.

A stress test consists of four parts:

1. The solution you want to test.

2. A different, possibly trivial and very slow, but easy to implement and obviously correct solution to the problem.

3. A test generator.

4. An infinite loop in which a new test is generated, it is fed into both solutions, then the results are compared, and if they differ, the test and both answers are output, and the program stops, otherwise the loop repeats.

The idea behind stress testing is that if you have two correct solutions, and the answer to the problem is unique, then for any possible test case they are guaranteed to give the same answer. If, however, at least one of the solutions is incorrect, then with very high probability there exists a test on which their answers differ. The only case when it is not so is when there is a common mistake in both solutions, but that is very unlikely (unless the mistake is somewhere in the input/output routines which are common to both solutions - check for that separately). Indeed, if one solution is correct and the other is wrong, then there obviously exists a test case on which they differ. If both are wrong, but the bugs are different, then most probably there exists a test on which one solution gives a correct answer and another gives wrong answer, so they differ.

Sample implementation

A sample implementation of the stress test for the maximum pairwise product problem can be found in the file below:

cpp_stress_test.cpp

This code is written on top of our last submission to the system. In addition to what was there - a slow solution and a fast solution - it has an `#include < cstdlib >` to use a function for generation of (pseudo-)random numbers and an infinite `while(true)` loop in the `main` function before the input/solve/output code:

```
while (true) {
  int n = rand() % 10 + 2;
  cerr << n << "\n";
  vector<int> a;
  for (int i = 0; i < n; ++i) {
    a.push_back(rand() % 100000);
  }
  for (int i = 0; i < n; ++i) {
    cerr << a[i] << ' ';
  }
  cerr << "\n";
  long long res1 = MaxPairwiseProduct(a);
  long long res2 = MaxPairwiseProductFast(a);
  if (res1 != res2) {
    cerr << "Wrong answer: " << res1 << ' ' << res2 << "\n";
    break;
  }
  else {
    cerr << "OK\n";
  }
}
```

This `while` loop starts with generating a number n - the length of the input sequence. n is generated as a random number between 2 and 11. It is at least 2, because the problem statement doesn't allow n to be less than 2. It is at most 11, so that it is not very big, because one of our solutions is usually slow, and we need it to run fast, so that we can check many different tests.

We then output n immediately, so that in the process of infinite loop we always know what is the size of the test on which we run our solutions, and also what is the test itself. This is important, because if you do a mistake in the test generation code, you will notice that easily and soon.

After we've generated n, we generate n random numbers in array a. Those are generated between 0 and 99999, because the problem statement only allows numbers not bigger than 100000. We could also generate numbers up to 100000 if that could matter. We then immediately output the array a in one line separated by spaces.

Now, we have a full test generated, and we run both our solutions on this test and save the results. We then compare the results. If they are different, we output "Wrong answer:" and both results and stop. Otherwise, we just output "OK" and go on.

We keep all the code which was in the `main()` function before, but it won't run until the stress test stops and finds a test for which our solutions differ, so it doesn't matter now.

Running the stress test

If we go on and compile our stress test using the settings given on in the reading "Available Programming Languages"

```
g++ -pipe -O2 -std=c++11
```

and run it, we will see a bunch of tests output on our screen with the word "OK" following each of them and then after a few dozen of tests we will see "Wrong answer" on the screen, and before that is the test on which our solutions give different results - 11 numbers, and those numbers are somewhat big - and the results of both solutions:

```
...
OK
3
67232 68874 69499
OK
8
6132 56210 45236 95361 68380 16906 80495 95298
OK
11
62180 1856 89047 36823 14251 8362 34171 93584 87362 83341 8784
OK
6
21468 16859 82178 70496 82939 44491
OK
11
68165 30342 87637 74297 2904 32873 86010 87637 66131 82858 82935
Wrong answer: 7680243769 7537658370
```

(Note that on your computer the result can be different because of different compiler which generates a different sequence of random numbers.) Hurrah! We've found a test case on which solutions differ, so now we can check which one is correct (or, better, which of them is wrong, because both of them can potentially be wrong, but not both are correct). Then we can debug that particular solution on this test case, find a bug, fix it and repeat the whole process.

Finding a small and simple test case

But before diving into that, I suggest that we first try to generate the smallest and easiest possible test case, so that debugging is faster, because generating tests automatically and running our stress test is easy, but debugging is hard. To do that, we return in the code and change the range for n from 2 - 11 to, for example, 2 - 5, and the range for the numbers from 0 - 99999 to, for example, 0 - 9. We then recompile and run our stress test. Now our program fails on the following test:

```
...
3
7 3 6
0 2
OK
5
2 9 3 1 9
Wrong answer: 81 27
```

Our slow solution gives answer 81 which is obviously correct ($9 \cdot 9 = 81$, all the other products are smaller). Our fast solution, however, gives result 27, which is wrong.

Debugging the solution

Now is the time to debug our fast solution. Let's check which two numbers did it determine to be the two maximums. To do that, let's add a debug output using cout before returning the answer from the function. We output the indices of the two maximums:

```
long long MaxPairwiseProductFast(const vector<int>& numbers) {
  int n = numbers.size();
  int max_index1 = -1;
  for (int i = 0; i < n; ++i)
    if ((max_index1 == -1) || (numbers[i] > numbers[max_index1]))
      max_index1 = i;
  int max_index2 = -1;
  for (int j = 0; j < n; ++j)
    if ((numbers[j] != numbers[max_index1]) && ((max_index2 == -1) ||
      (numbers[j] > numbers[max_index2])))
      max_index2 = j;
  cout << max_index1 << ' ' << max_index2 << "\n";
  return ((long long)(numbers[max_index1])) * numbers[max_index2];
}

```

After we recompile and run, we see an additional line with the indices of the two maximums.

```
...
3
7 3 6
0 2
OK
5
2 9 3 1 9
1 2
Wrong answer: 81 27
```

Please note that our solutions worked and then failed on exactly the same test cases as on the previous run of the stress test, because we didn't change anything in the test generator, and the numbers it uses to generate tests are, although random, actually pseudorandom - it means that the sequence looks random, but it is the same each time we run this program. It is a very convenient and important property, and you should always try to have your programs exhibit such behavior, because deterministic programs (that always give the same result for the same input) are much easier to debug than non-deterministic ones.

Now let's see what are those indices - 1 and 2. Remember that the positions in the array in C++ are numbered from 0, so those are the second (9) and the third (3) number in the input. It means that `max_index1` was computed correctly, but `max_index2` was not. If we look in the code for determining the second maximum, we will notice after some thinking a subtle bug: when we implemented a condition on j, such that it is not the same as the previous maximum, instead of comparing j and `max_index1`, we compared `numbers[j]` with `numbers[max_index1]`. This ensured that the second maximum will differ from the first maximum by value, always. So, our solution fails on any test case where the two maximum numbers are equal. We now change the condition from

```
numbers[j] != numbers[max_index1]
```

to

```
1  j != max_index1
```

- now the condition is correct. Then we recompile and run the stress test. And now we see a screen that is constantly filled with test cases and words "OK" after them, but it doesn't stop. We wait for 10 or 30 seconds until we get bored and then decide that probably the solutions don't differ.

More thorough stress testing

However, we shouldn't stop here. We should remember that we now only generate very small tests with n from 2 to 5 and numbers in the input from 0 to 9. We should check whether our solutions work well for bigger numbers and bigger size of the input. So, we change the restriction for n to the range from 2 to 1000 (for bigger n, slow solution will be pretty slow, because its running time is quadratic). We also change the range for numbers to 0 - 99999, recompile and run. We again see the screen filling with much bigger tests, words "OK", but the program doesn't stop. After we wait for some time, we think that our solution now works correctly.

Prepare for submission

Now let's convert it to a submission. We just comment out the whole `while(true)` loop that implements stress test

```
int main() {
/*  while (true) {
    int n = rand() % 1000 + 2;
    cerr << n << "\n";
    vector<int> a;
    for (int i = 0; i < n; ++i) {
      a.push_back(rand() % 100000);
    }
    for (int i = 0; i < n; ++i) {
      cerr << a[i] << ' ';
    }
    cerr << "\n";
    long long res1 = MaxPairwiseProduct(a);
    long long res2 = MaxPairwiseProductFast(a);
    if (res1 != res2) {
      cerr << "Wrong answer: " << res1 << ' ' << res2 << "\n";
      break;
    }
    else {
      cerr << "OK\n";
    }
  }*/
  int n;
  cin >> n;
  vector<int> numbers(n);
  for (int i = 0; i < n; ++i) {
    cin >> numbers[i];
  }

  long long result = MaxPairwiseProductFast(numbers);
  cout << result << "\n";
  return 0;
}

```

, recompile and run. Now our solution waits for us to input a test. We input a test and see the answer:

```
./stress_test
2
3 5
0 1
15
```

This is not what we expected! 15 is the expected answer, but what are 0 and 1? Well, it's our debug output. It's a very common mistake to forget to comment out all the debug output before submitting the solution. If we submitted this code without testing it, we would get incorrect result on one of the first tests, because besides correct answer we output something else, which is not expected. Remember to always output only what you are asked in the problem statement, and nothing more, otherwise your result will be judged as incorrect. So, we go back to the code, comment out the debug output

```
...
  //cout << max_index1 << ' ' << max_index2 << "\n";

  return ((long long)(numbers[max_index1])) * numbers[max_index2];
...
```

, recompile and run on the same test

```
./stress_test
2
3 5
15
```

Now our solution just outputs "15", so we are ready to submit.

Submit the fixed solution

We go on to submit it in the system, wait for some time...and it passes!

cpp_stress_test_final.cpp

Further advice

This illustrates the stress testing technique, but let us give you some additional advice.

First, you noticed that the stress test "magically" gave us the test on which our solutions differ from the first attempt. This is not always like that. For example, we know that in this problem our initial solutions differed only in the case when the two maximal numbers in the input were the same. This is a pretty rare case if we generate lots of big numbers. So, we were lucky to find such a case when we generated numbers from 0 to 99999. If we started with generating numbers from a smaller range, we would find the test like that much faster, because number would coincide much more often.

This is true also in the general case: when you do stress testing, often the case with equal numbers or equal letters in a string, etc. is a kind of corner case. To ensure that your stress test finds that fast enough for you to notice, you should try to not only generate just some random test cases, but also try to generate some more "focused" random test cases, such as those with only small numbers or a small range of big numbers, only strings which consist of only letter "a" or only of two different letters (as opposed to strings composed of all possible lowercase latin letters), and so on. Think about other subspaces of possible tests which could be peculiar in some sense. For example, if you are generating graphs, try generating trees, disconnected graphs, full graphs, bipartite graphs. If you generate trees, try generating chains, balanced binary trees, "stars" - trees with a root and many direct children. If you are generating integer numbers, try generating prime and composite numbers. If you are generating sets of points, try generating them on one line and on one circle, etc.

Another important point is to try generating the smallest possible test case for which one of the solutions fails: it will simplify debugging a lot.

Don't expect stress testing to be the silver bullet. For example, it won't detect the problems your solution has with time limit or memory limit, because you will generate smaller tests because of the second, "trivial" solution, which would run too slow on those. Also, you probably won't detect the

integer overflow problems, because you'll generate only small numbers, or because you'll have integer overflow problems in both solutions. You should really first test for all of those separately, before implementing a stress test.

However, if you've tried all of those tests listed in the beginning, stress test should be your next move. It is almost guaranteed to find you the test case on which your solution fails, especially if you try generating tests cleverly, in different subspaces of the test space, trying to make equal numbers or equal letters more likely, and trying to make other rare cases appear more often in the generated tests. The reason is that when the authors of the problems generate test suites, they follow more or less the same process. They add manual tests, corner case tests, big tests, integer overflow tests, and then they implement a few test generators and add them. They also implement wrong solutions which are expected from the learners and run the generators until they find the tests on which they differ from the correct solution, then they add those tests to the suite. So, if you follow the same process, although it is somewhat randomized, you are very likely to find all the traps they've set up for you :)

Conclusion

Note that it is very important to learn to write programs that work correctly on all the tests which are allowed by the problem statement, and not just on most of them. In practice, doing otherwise leads to bugs in the services, and when users encounter your bugs, their experience deteriorates significantly. When you make mistakes that only rarely reveal themselves, those can affect the whole project: you will get some results of testing your hypotheses and will believe in them for a long time, base your decisions on that belief, only to learn later that the code had a bug in it. That is why we've prepared thorough test suites for all the problems that will try to check all the possible aspects where your solution might fail. And that is where stress testing will be so helpful - when you're stuck with a problem and don't even know what is the test case on which it fails. We hide it for purpose, so that you learn to find this test case yourself: in the real life very often you won't know what are the exact conditions under which your program fails.

Key take-aways

1. It is very important to write programs that work correctly on all the allowed inputs.
2. Testing is essential to writing correct programs.
3. First test on a few small manual tests, then test for each type of answer, then test on large test cases for time limit and memory limit, then test on corner cases.
4. After that, apply stress testing to ensure your program works - it will almost always lead to correct solution. You can do it before your first attempt to submit your solution - and will often get it right from the first attempt!
5. Stress testing consists of implementing the intended solution, another simple possible slow solution, a test generator and an infinite loop which generates tests and compares answers of the two solutions.
6. Always try to find the smallest test cases on which your solution fails.

7. Try different regions of the test space when generating cases for stress testing.

✓ Complete