Introduction and Learning Outcomes

In this sequence of readings, we show a sample process of implementing and testing your solution for the maximum pairwise product problem.

In this reading you will ...

1. Download a starter solution, then compile and run it.
2. Submit it to the testing system to find out that it is buggy.
3. Use the feedback message from the testing system to fix an integer overflow bug.
4. Submit the solution to the testing system again to figure out that it is too slow.
5. Implement a new faster algorithm.

Running a Starter Solution

Assume that you are going to use the C++ programming language to solve this assignment. You then first download a template solution file `max_pairwise_product.cpp`:

```cpp
#include <iostream>
#include <vector>

using std::vector;
using std::cin;
using std::cout;

int MaxPairwiseProduct(const vector<int>& numbers) {
  int result = 0;
  int n = numbers.size();
  for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
      if (numbers[i] * numbers[j] > result) {
        result = numbers[i] * numbers[j];
      }
    }
  }
  return result;
}

int main() {
    int n;
    cin >> n;
    vector<int> numbers(n);
    for (int i = 0; i < n; ++i) {
        cin >> numbers[i];
    }

    int result = MaxPairwiseProduct(numbers);
    cout << result << "\n";
    return 0;
}
```

You compile it using the same flags as on the testing machine:

```
g++ -pipe -O2 -std=c++11 max_pairwise_product.cpp

```

This produces an executable file (either `a.out` or `a.exe`, depending on your operating system). You run the program on a small sequence:

```
./a.out
3
7 2 5
```

The program outputs the correct answer:

```
35

```

Now you are curious what you are supposed to do in this problem. Indeed, the template solution contains a program that solves the problem correctly, for an obvious reason: to find a pairwise maximum product, it just goes through all possible pairwise products and selects the largest one. There are two hidden dangers here, however.

Submitting to the Testing System

You encounter the first pitfall by submitting this file as a solution to the corresponding problem. Surprisingly, you get the following feedback from the testing system:

```
Failed case #3 (Wrong answer)
Input:
2
100000 90000
Your output:410065408

Correct output:9000000000
```

Fixing an Integer Overflow Bug

By looking at this example, you realize that the reason of this failure is an integer overflow. Indeed such a large number as $9,000,000,000$ does not fit into the standard C++ **`int`** type (on most modern machines, it ranges from $-2,147,483,648$ to $2,147,483,647$). You then adjust the type of the variable result to long long as follows (this requires to change five lines of code!):

```
#include <iostream>
#include <vector>

using std::vector;
using std::cin;
using std::cout;

long long MaxPairwiseProduct(const vector<int>& numbers) {
  long long result = 0;
  int n = numbers.size();
  for (int i = 0; i < n; ++i) {
    for (int j = i + 1; j < n; ++j) {
      if (((long long)numbers[i]) * numbers[j] > result) {
        result = ((long long)numbers[i]) * numbers[j];
      }
    }
  }
  return result;
}

int main() {
    int n;
    cin >> n;
    vector<int> numbers(n);
    for (int i = 0; i < n; ++i) {
        cin >> numbers[i];
    }

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

This is already safe: the long long type ranges from $-9,223,372,036,854,775,807$ to $9,223,372,036,854,775,807$, while in our problem all the numbers in the sequence are non-negative integers not exceeding $100,000$ so the product of two such numbers is an integer between 0 and $10,000,000,000$.

Now, you are completely sure that the algorithm is correct: it correctly solves the problem and uses an appropriate type for storing the result. You submit it to the testing system.

Implementing a Faster Solution

Surprisingly, it fails again! Now the error message says

```
Failed case #4: time limit exceeded
```

This is because our program performs about n^2 steps on a sequence of length n. For the maximal possible value $n = 200,000 = 2 \cdot 10^5$, the number of steps is about $10,000,000,000 = 10^{10}$. This is too much. Recall that modern machines can perform roughly 10^9 basic operations per second (this depends on a machine of course, but 10^9 is a reasonable average estimate). Thus, we need a faster algorithm.

In search of an inspiration, you start to play with small examples. How to find the maximal pairwise product of a sequence 1,2,3,4? Well, of course, it suffices to multiply the two largest numbers — 3 and 4. And this is true in general since all numbers in our sequence are non-negative.

You implement the algorithm that you have just discovered as follows. For testing purposes, you also keep the previous algorithm.

```
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;

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

Testing

You also change the main function as follows:

```
int main() {
  int n;
  cin >> n;
  vector<int> numbers(n);
  for (int i = 0; i < n; ++i) {
    cin >> numbers[i];
  }

  long long result1 = MaxPairwiseProduct(numbers);
  long long result2 = MaxPairwiseProductFast(numbers);
  cout << result1 << "\n" << result2;
  return 0;
}
```

You compile your new program and run it on the previous small example:

```
./a.out
3
7 2 5
```

Quite satisfactory, it outputs the right answer two times:

```
35
35

```

Now, you decide to check how long does it take your program to process a large dataset. For this, you pass an array of size $200,000 = 2 \cdot 10^5$ filled in by zeroes to your new function **`MaxPairwiseProductFast`**:

```
long long result = MaxPairwiseProductFast(vector<int>(200000, 0));
cout << result << "\n";
```

It outputs 0 just immediately which shows that your new algorithm is indeed fast. Thus, your new solution is both fast and correct. This is exactly what is needed to solve the problem. Inspired by this fact, you submit the solution to the testing system again. And guess what? It fails once again!!

```
Failed case #5: incorrect result
```

An important remark. When testing your program on large data sets, it makes sense to first generate a data set and to store it in a file and then to redirect its contents to the standard input when running your program:

```
generate_test > test.txt
your_program < test.txt
```

The reason for this is straightforward: for large data sets, your program may spend a noticeable amount of time already for reading the input data.

What's Up Next?

In the next screencast/reading we will learn how to use stress testing to locate a bug in the fast algorithm.

✓ Complete