2 Maximum Pairwise Product

Maximum Pairwise Product Problem
Find the maximum product of two distinct numbers in a sequence of non-negative integers.

Input: A sequence of non-negative integers.
Output: The maximum value that can be obtained by multiplying two different elements from the sequence.

	5	6	2	7	4
5		30	10	35	20
6	30		12	42	24
2	10	12		14	8
7	35	42	14		28
4	20	24	8	28	

Given a sequence of non-negative integers $a_1, \ldots, a_n$, compute

$$\max_{1 \le i \ne j \le n} a_i \cdot a_j .$$

Note that i and j should be different, though it may be the case that $a_i = a_j$.

Input format. The first line contains an integer n. The next line contains n non-negative integers $a_1, \ldots, a_n$ (separated by spaces).

Output format. The maximum pairwise product.

Constraints. $2 \le n \le 2 \cdot 10^5$; $0 \le a_1, \ldots, a_n \le 2 \cdot 10^5$.

Sample 1.

Input:

```
3
1 2 3
```

Output:

```
6
```

Sample 2.

Input:

```
10
7 5 14 2 8 8 10 1 2 3
```

Output:

```
140
```

Time and memory limits. The same as for the previous problem.

2.1 Naive Algorithm

A naive way to solve the Maximum Pairwise Product Problem is to go through all possible pairs of the input elements $A[1 \dots n] = [a_1, \dots, a_n]$ and to find a pair of distinct elements with the largest product:

```
MaxPairwiseProductNaive(A[1...n]):
product ← 0
for i from 1 to n:
  for j from 1 to n:
    if i ≠ j:
      if product < A[i] · A[j]:
        product ← A[i] · A[j]
return product
```

This code can be optimized and made more compact as follows.

```
MaxPairwiseProductNaive(A[1...n]):
product ← 0
for i from 1 to n:
  for j from i + 1 to n:
    product ← max(product, A[i] · A[j])
return product
```

Implement this algorithm in your favorite programming language. If you are using C++, Java, or Python3, you may want to download the starter files (we provide starter solutions in these three languages for all the problems in the book). For other languages, you need to implement your solution from scratch.

Starter solutions for C++, Java, and Python3 are shown below.

C++

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

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

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

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

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

Java

```
import java.util.*;
import java.io.*;
```

```
public class MaxPairwiseProduct {
    static int getMaxPairwiseProduct(int[] numbers) {
        int product = 0;
        int n = numbers.length;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                product = Math.max(product,
                                   numbers[i] * numbers[j]);
            }
        }
        return product;
    }

    public static void main(String[] args) {
        FastScanner scanner = new FastScanner(System.in);
        int n = scanner.nextInt();
        int[] numbers = new int[n];
        for (int i = 0; i < n; i++) {
            numbers[i] = scanner.nextInt();
        }
        System.out.println(getMaxPairwiseProduct(numbers));
    }

    static class FastScanner {
        BufferedReader br;
        StringTokenizer st;

        FastScanner(InputStream stream) {
            try {
                br = new BufferedReader(new
                    InputStreamReader(stream));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        String next() {
            while (st == null || !st.hasMoreTokens()) {
                try {
```

```
                st = new StringTokenizer(br.readLine());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return st.nextToken();
    }

    int nextInt() {
        return Integer.parseInt(next());
    }
  }

}
```

Python

```
# Uses python3
n = int(input())
a = [int(x) for x in input().split()]

product = 0

for i in range(n):
    for j in range(i + 1, n):
        product = max(product, a[i] * a[j])

print(product)
```

After submitting this solution to the grading system, many students are surprised when they see the following message:

```
Failed case #4/17: time limit exceeded
```

After you submit your program, we test it on dozens of carefully designed test cases to make sure the program is fast and error proof. As the result, we usually know what kind of errors you made. The message above tells that the submitted program exceeds the time limit on the 4th out of 17 test cases.

Stop and Think. Why does the solution not fit into the time limit?

MaxPairwiseProductNaive performs of the order of n^2 steps on a sequence of length n. For the maximal possible value $n = 2 \cdot 10^5$, the number of steps is of the order $4 \cdot 10^{10}$. Since many modern computers perform roughly 10^8–10^9 basic operations per second (this depends on a machine, of course), it may take tens of seconds to execute MaxPairwiseProductNaive, exceeding the time limit for the Maximum Pairwise Product Problem.

We need a faster algorithm!

2.2 Fast Algorithm

In search of a faster algorithm, you play with small examples like $[5, 6, 2, 7, 4]$. Eureka—it suffices to multiply the two largest elements of the array—7 and 6!

Since we need to find the largest and the second largest elements, we need only two scans of the sequence. During the first scan, we find the largest element. During the second scan, we find the largest element among the remaining ones by skipping the element found at the previous scan.

```
MaxPairwiseProductFast(A[1...n]):
index_1 ← 1
for i from 2 to n:
  if A[i] > A[index_1]:
    index_1 ← i
index_2 ← 1
for i from 2 to n:
  if A[i] ≠ A[index_1] and A[i] > A[index_2]:
    index_2 ← i
return A[index_1] · A[index_2]
```

2.3 Testing and Debugging

Implement this algorithm and test it using an input $A = [1, 2]$. It will output 2, as expected. Then, check the input $A = [2, 1]$. Surprisingly, it outputs 4. By inspecting the code, you find out that after the first loop, $index_1 = 1$. The algorithm then initializes $index_2$ to 1 and $index_2$ is never

updated by the second for loop. As a result, $index_1 = index_2$ before the return statement. To ensure that this does not happen, you modify the pseudocode as follows:

```
MaxPairwiseProductFast(A[1...n]):
index_1 ← 1
for i from 2 to n:
  if A[i] > A[index_1]:
    index_1 ← i
if index_1 = 1:
  index_2 ← 2
else:
  index_2 ← 1
for i from 1 to n:
  if A[i] ≠ A[index_1] and A[i] > A[index_2]:
    index_2 ← i
return A[index_1] · A[index_2]
```

Check this code on a small datasets $[7, 4, 5, 6]$ to ensure that it produces correct results. Then try an input

```
2
100000  90000
```

You may find out that the program outputs something like 410 065 408 or even a negative number instead of the correct result 9 000 000 000. If it does, this is most probably caused by an *integer overflow*. For example, in C++ programming language a large number like 9 000 000 000 does not fit into the standard `int` type that on most modern machines occupies 4 bytes and ranges from -2^{31} to $2^{31} - 1$, where

$$2^{31} = 2\,147\,483\,648.$$

Hence, instead of using the C++ `int` type you need to use the `int64_t` type when computing the product and storing the result. This will prevent integer overflow as the `int64_t` type occupies 8 bytes and ranges from -2^{63} to $2^{63} - 1$, where

$$2^{63} = 9\,223\,372\,036\,854\,775\,808.$$

You then proceed to testing your program on large data sets, e.g., an array $A[1 \dots 2 \cdot 10^5]$, where $A[i] = i$ for all $1 \le i \le 2 \cdot 10^5$. There are two ways of doing this.

1. Create this array in your program and pass it to MaxPairwiseProductFast (instead of reading it from the standard input).

2. Create a separate program, that writes such an array to a file `dataset.txt`. Then pass this dataset to your program from console as follows:

```
yourprogram < dataset.txt
```

Check that your program processes this dataset within time limit and returns the correct result: 39 999 800 000. You are now confident that the program finally works!

However, after submitting it to the testing system, it fails again...

```
Failed case #5/17: wrong answer
```

But how would you generate a test case that make your program fail and help you to figure out what went wrong?

2.4 Can You Tell Me What Error Have I Made?

You are probably wondering why we did not provide you with the 5th out of 17 test datasets that brought down your program. The reason is that nobody will provide you with the test cases in real life!

Since even experienced programmers often make subtle mistakes solving algorithmic problems, it is important to learn how to catch bugs as early as possible. When the authors of this book started to program, they naively thought that nearly all their programs are correct. By now, we know that our programs are *almost never* correct when we first run them.

When you are confident that your program works, you often test it on just a few test cases, and if the answers look reasonable, you consider your work done. However, this is a recipe for a disaster. To make your program *always* work, you should test it on a set of carefully designed test cases. Learning how to implement algorithms as well as test and debug your programs will be invaluable in your future work as a programmer.

2.5 Stress Testing

We will now introduce *stress testing*—a technique for generating thousands of tests with the goal of finding a test case for which your solution fails.

A stress test consists of four parts:

1. Your implementation of an algorithm.
2. An alternative, trivial and slow, but correct implementation of an algorithm for the same problem.
3. A random test generator.
4. An infinite loop in which a new test is generated and fed into both implementations to compare the results. If their results differ, the test and both answers are output, and the program stops, otherwise the loop repeats.

The idea behind stress testing is that two correct implementations should give the same answer for each test (provided the answer to the problem is unique). If, however, one of the implementations is incorrect, then there exists a test on which their answers differ. The only case when it is not so is when there is the same mistake in both implementations, but that is unlikely (unless the mistake is somewhere in the input/output routines which are common to both solutions). Indeed, if one solution is correct and the other is wrong, then there exists a test case on which they differ. If both are wrong, but the bugs are different, then most likely there exists a test on which two solutions give different results.

Here is the the stress test for MaxPairwiseProductFast using MaxPairwiseProductNaive as a trivial implementation:

```
StressTest(N, M):
while true:
  n ← random integer between 2 and N
  allocate array A[1...n]
  for i from 1 to n:
    A[i] ← random integer between 0 and M
  print(A[1...n])
  result_1 ← MaxPairwiseProductNaive(A)
  result_2 ← MaxPairwiseProductFast(A)
  if result_1 = result_2:
    print("OK")
  else:
    print("Wrong answer: ", result_1, result_2)
    return
```

The `while` loop above starts with generating the length of the input sequence n, a random number between 2 and N. It is at least 2, because the problem statement specifies that $n \geq 2$. The parameter N should be small enough to allow us to explore many tests despite the fact that one of our solutions is slow.

After generating n, we generate an array A with n random numbers from 0 to M and output it so that in the process of the infinite loop we always know what is the current test; this will make it easier to catch an error in the test generation code. We then call two algorithms on A and compare the results. If the results are different, we print them and halt. Otherwise, we continue the while loop.

Let's run StressTest(10, 100 000) and keep our fingers crossed in a hope that it outputs "Wrong answer." We see something like this (the result can be different on your computer because of a different random number generator).

```
...
OK
67232 68874 69499
OK
6132 56210 45236 95361 68380 16906 80495 95298
OK
62180 1856 89047 14251 8362 34171 93584 87362 83341 8784
OK
```

```
21468 16859 82178 70496 82939 44491
OK
68165 87637 74297 2904 32873 86010 87637 66131 82858 82935
Wrong answer: 7680243769 7537658370
```

Hurrah! We've found a test case where MaxPairwiseProductNaive and MaxPairwiseProductFast produce different results, so now we can check what went wrong. Then we can debug this solution on this test case, find a bug, fix it, and repeat the stress test again.

Stop and Think. Do you see anything suspicious in the found dataset?

Note that generating tests automatically and running stress test is easy, but debugging is hard. Before diving into debugging, let's try to generate a smaller test case to simplify it. To do that, we change N from 10 to 5 and M from 100 000 to 9.

Stop and Think. Why did we first run StressTest with large parameters N and M and now intend to run it with small N and M?

We then run the stress test again and it produces the following.

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

The slow MaxPairwiseProductNaive gives the correct answer 81 ($9 \cdot 9 = 81$), but the fast MaxPairwiseProductFast gives an incorrect answer 27.

Stop and Think. How MaxPairwiseProductFast can possibly return 27?

To debug our fast solution, let's check which two numbers it identifies as two largest ones. For this, we add the following line before the `return` statement of the MaxPairwiseProductFast function:

```
print(index1, index2)
```

After running the stress test again, we see the following.

```
...
7 3 6
1 3
```

```
OK
5
2 9 3 1 9
2 3
Wrong answer: 81 27
```

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. The numbers it uses to generate tests are pseudorandom rather than random—it means that the sequence looks random, but it is the same each time we run this program. It is a convenient and important property, and you should try to have your programs exhibit such behavior, because deterministic programs (that always give the same result for the same input) are easier to debug than non-deterministic ones.

Now let's examine $index_1 = 2$ and $index_2 = 3$. If we look at the code for determining the second maximum, we will notice a subtle bug. When we implemented a condition on i (such that it is not the same as the previous maximum) instead of comparing i and $index_1$, we compared $A[i]$ with $A[index_1]$. This ensures that the second maximum differs from the first maximum by the value rather than by the index of the element that we select for solving the Maximum Pairwise Product Problem. So, our solution fails on any test case where the largest number is equal to the second largest number. We now change the condition from

$$A[i] \neq A[index_1]$$

to

$$i \neq index_1$$

After running the stress test again, we see a barrage of "OK" messages on the screen. We wait for a minute until we get bored and then decide that MaxPairwiseProductFast is finally correct!

However, you shouldn't stop here, since you have only generated very small tests with $N = 5$ and $M = 10$. We should check whether our program works for larger n and larger elements of the array. So, we change N to 1 000 (for larger N, the naive solution will be pretty slow, because its running time is quadratic). We also change M to 200 000 and run. We again see the screen filling with words "OK", wait for a minute, and then

decide that (finally!) MaxPairwiseProductFast is correct. Afterwards, we submit the resulting solution to the grading system and pass the Maximum Pairwise Product Problem test!

As you see, even for such a simple problems like Maximum Pairwise Product, it is easy to make subtle mistakes when designing and implementing an algorithm. The pseudocode below presents a more "reliable" way of implementing the algorithm.

```
MaxPairwiseProductFast(A[1...n]):
index ← 1
for i from 2 to n:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n]
index ← 1
for i from 2 to n − 1:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n − 1]
return A[n − 1] · A[n]
```

In this book, besides learning how to design and analyze algorithms, you will learn how to implement algorithms in a way that minimizes the chances of making a mistake, and how to test your implementations.

2.6 Even Faster Algorithm

The MaxPairwiseProductFast algorithm finds the largest and the second largest elements in about $2n$ comparisons.

Exercise Break. Find two largest elements in an array in $1.5n$ comparisons.

After solving this problem, try the next, even more challenging Exercise Break.

Exercise Break. Find two largest elements in an array in $n + \lceil \log_2 n \rceil - 2$ comparisons.