Competetive Programming Cookbook

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

1 Getting started 2

1.1 Input and output 2

1.2 Working with numbers 3

1.3 Modular arithmetic 3

Getting started

1.1 Input and output

In most contests, standard streams are used for reading input and writing output. In C++, the standard streams are cin for input and cout for output. In addition, the C functions scanf and printf can be used.

Input and output is sometimes a bottleneck in the program. The following lines at the beginning of the code make input and output more efficient

```
ios::sync_with_stdio(0);
cin.tie(0);
```

- **ios::sync_with_stdio(0);**: This disables the synchronization between C++ standard streams (like cin and cout) and C standard streams (like scanf and printf). Without synchronization, C++ I/O becomes faster, but you should avoid mixing C++ and C I/O operations.
- **cin.tie(0);**: By default, cin is tied to cout, which means that cout is automatically flushed before any input operation to ensure correct sequencing. Untying them (by setting the tie to nullptr or 0) stops this automatic flush, which can improve performance when you don't need the interleaving of output with input.

Note that the newline "" works faster than endl, because endl always causes a flush operation

The C functions scanf and printf are an alternative to the C++ standard streams. They are usually a bit faster, but they are also more difficult to use. The following code reads two integers from the input:

```
int a,b;
scanf("%d %d", &a, &b);
```

The following code prints two integers:

```
printf("%d %d\n", a,b);
```

In some contest systems, files are used for input and output. An easy solution for this is to write the code as usual using standard streams, but add the following lines to the beginning of the code:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

After this, the program reads the input from the file "input.txt" and writes the output to the file "output.txt"

1.2 Working with numbers

The most used integer type in competitive programming is int, which is a 32-bit type with a value range of -2^{31} to $2^{31}-1$ or about $-2 \cdot 10^9$ to $2 \cdot 10^9$. If the type int is not enough, the 64-bit type long long can be used. It has a value range of -2^{63} to $2^{63}-1$ or about $-9 \cdot 10^{18}$ to $9 \cdot 10^{18}$

```
long long x = 123456789123456789LL;
```

The suffix LL means that the type of the number is long long.

A common mistake when using the type long long is that the type int is still used somewhere in the code. For example, the following code contains a subtle error

```
int a = 123456789;
long long b = a*a;
cout << b << "\n"; // -1757895751
```

Even though the variable b is of type long long, both numbers in the expression a*a are of type int and the result is also of type int. Because of this, the variable b will contain a wrong result. The problem can be solved by changing the type of a to long long or by changing the expression to (long long)a*a.

Usually contest problems are set so that the type long long is enough. Still, it is good to know that the g++ compiler also provides a 128-bit type __int128_t with a value range of -2^{127} to $2^{127}-1$ or about -10^{38} to 10^{38}. However, this type is not available in all contest systems.

1.3 Modular arithmetic

We denote by $x \mod m$ the remainder when x is divided by m. For example, $17 \mod 5 = 2$, because $17 = 3 \text{\r{u}} 5 + 2$.

Sometimes, the answer to a problem is a very large number but it is enough to output it "modulo m", i.e., the remainder when the answer is divided by m example, "modulo $10^9 + 7$"). The idea is that even if the actual answer is very large, it suffices to use the types int and long long.

An important property of the remainder is that in addition, subtraction and multiplication, the remainder can be taken before the operation:

$$\begin{aligned}
(a+b) \mod m &= (a \mod m + b \mod m) \mod m \\
(a-b) \mod m &= (a \mod mb \mod m) \mod m \\
(a \cdot b) \mod m &= (a \mod m \text{\r{u}} b \mod m) \mod m
\end{aligned}$$

Thus, we can take the remainder after every operation and the numbers will never become too large.

Usually we want the remainder to always be between 01. However, in C++ and other languages, the remainder of a negative number is either zero or negative. An easy way to make sure there are no negative remainders is to first calculate the remainder as usual and then add m if the result is negative:

```
x = x%m;
if (x < 0) x += m;
```

However, this is only needed when there are subtractions in the code and the remainder may become negative.