Tips and Tricks for Beginners in Competitive Programming

Peter Rossmanith

September 1, 2022

Abstract

Taking part in competitive programming contest presents several challenges for students even if they have training in programming and algorithm design. We discuss several of the problems and how to address them. This includes among others input/output handling, number overflows, chosing the right programming language, and how to train for competitions.

1 Who should read this document?

If you are already familiar with programming contests in the ICPC of IOI style, you will probably not find much useful information here. If you are a student at RWTH Aachen and have some computer science experience in the form of a programming lecture and a lecture on data structures and algorithm, have not taken part in programming contests, but plan to do so in the future, please read on.

2 Contest Format and Rules

In a programming contest under ICPC rules teams of three persons are gathering in a large room. Every team has a desk and a computer with limited capabilies: You cannot access the internet freely and the installed software consists only of the necessary compilers and IDE's for the allowed programming languages. In particular you cannot lookup algorithms on the internet and you cannot even lookup language features. You can, however, bring *printed* material to the contest. In some online contests looking up information on the internet is allowed as long as the information existed before the contest started.

Of course, you are allowed access the judge server in order to submit your programs and to view the results. It is important to try out the judge system before a real contest. Practice to submit a program. It is very easy, but you do not want to waste time finding out where the right menu etc. is.

The winners are determined as follows: The most important criterion is how many tasks are successfully solved. If a team submits correct solution in the last minute for the five most easy tasks and another team in the first minute for the four hardest tasks, the team with the five correct tasks is better. If two teams solve the same number of tasks, speed is important, i.e., the total time for solving the tasks. If you submit a wrong solution you also get a penalty of 10 minutes (depends on the contest, but we have it), so be a bit careful before you submit it. The penalty is only applied if you ultimately solve the task. It is not applied if all submissions to the task are wrong. In particular you should test your program on the sample instances and maybe on some additional instances that you prepare yourself.

3 Input and Output

In programming contest you will usually not use files for input or output. You read from standard input and write to standard output. You might not be used to this type of I/O and it very important that you learn to use it properly.

Most of the time the input consists of several "parts," which are typically numbers or strings. These individual items are usually separated by white space, i.e., spaces, tabs and new lines. The input format specification is usually quite specific and tells you what to expect in each line of the input. Let us look at an example that is *not* that specific: The input is a number n followed by n numbers $a_1, \ldots, a_n$. As a result you should print their sum $a_1 + \cdots + a_n$. Let us assume that it is allowed that all n numbers appear in a single line, but also that several numbers appear in a line, but then a new line follows whenever the line becomes too long. How can you solve this task? Most easily you just read along not caring what white space symbols are separating the numbers. Here is an example how to do this in C:

```
int main() {
  int n, sum = 0;
  for(int i=1; i<=n; i++) {
    int x; scanf("%d", &x);
    sum += x;
  }
  printf("%d\n", sum);
```

```
}
```

The solution is quite simple because `scanf` just reads the next integer skipping any white space. In C++ it is basically the same game:

```
int main() {
  int n, sum = 0;
  cin >> n;
  for(int i=1; i<=n; i++) {
    int x; cin >> x;
    sum += x;
  }
  cout << sum << "\n";
}
```

These languages have a built in mechanism to read integers and also strings, which are not part of this example.

In Java, use can use the class `Scanner` to achieve the same effect.

```
import java.util.Scanner;
public class Add {
  public static void main(String args[]) {
    Scanner s = new Scanner(System.in);
    int n = s.nextInt(), sum = 0;
    for(int i=1; i<=n; i++) {
      sum += s.nextInt();
    }
    System.out.println(sum);
  }
}
```

Python seems to be line oriented and I do not know how to achieve the same effect. My solution is to program a function myself that reads the next white space separated item as a string:

```
toks = []
def next():
    global toks
    if toks == []:
        toks = input().split()
        toks.reverse()
    return toks.pop()
```

```
n = int(next())
total = 0
for i in range(n):
  total += int(next())
print(total)
```

All these programs are sufficiently fast for our purposes and work however the numbers are separated.

EOF. Traditionally the input format in programming contest is given in such way that you easily can find out when the necessary information has been read *without testing for end of file.* When you read from a file it will end eventually because files have a finite length. You can (and in contest systems this happens all the time), however, also read from a pipe or a socket, where the input stream might never end. A pipe can also become empty and the writer on the other end just stops sending data without closing the pipe. If you wait in such a situation for EOF, your process will just be blocked forever and in a contest you will get a time-out.

It is best not to wait for EOF, but use the self limiting nature of the input format to determine when to stop. Look at the examples above. They do not wait for EOF.

Flushing. If you print a lot of output, you should not flush stdout very often. Flushing an output file means telling the operating system that you would like to wait until the output operation has physically ended. In case of a file on disk this means waiting until the data has been written to the disk, which takes a long time. While the example with the disk is extreme, flushing always means waiting and you do not want to wait. In C++ you should not use `endl` because it automatically flushes stdout. Use `'\n'` instead.

There are times, however, when flushing stdout is necessary. This happens in interactive tasks, where you have to react often and immeadiate. Let us say your input consists of n followed by n numbers $a_1, \ldots, a_n$ and after reading a_i you have to output $\max\{a_1, \ldots, a_i\}$, the maximum of the numbers read so far. The system will give you a_{i+1} only after you it has received your answer to a_i. If you write that answer to stdout without flushing, the whole system will be blocked and you lose with a timeout. You can flush stdout in C with `fflush(stdout)`, in C++ you can use `std::cout << std::flush`, in Java `System.out.flush()`, and in Python `sys.stdout.flush()`. There are also other ways of flushing, these are some examples that should work.

In a nutshell: Do not flush unless in an interactive task.

Defensive parsing and paranoid printing. You are safest if you parse the input without assuming too much about its format, which you usually know from the task description and by looking at the sample inputs. Be defensive. For example, at the end of the lines there might be additional white space, which is not mentioned in the description and also not visible in the samples.

On the other hand, try to obey the instructions for the output format to the letter. Do not add extra empty lines or other white space. If the number format is specified, stick to it. In general do not assume that the judging system is lenient when parsing your output.

Testing your program. As your program reads from stdin it might be challenge for you to test it yourself if you have never used I/O redirection. An easy way to test such a program is to prepare a file with sample input. In a shell you can test the program by typing

```
./program < sample
```

if your program name is `program` and the file name with the input data is `sample`. The `<` tells the system to redirect the file content to the stdin of the running program. Test your program *at least* on the provided samples. If it fails this test it is definitely wrong.

4 Number formats and overflows

Very often you are handling numbers. Let us start with integers. The input format description usually tells you how big the numbers in the input can become. Values that you will see often are $\pm 10^5$, $\pm 10^6$, $\pm 10^9$, $\pm 10^{18}$ and also much smaller numbers. The problem, of course, are big numbers because they can overflow and give you a wrong result. Most programming languages have integer variables with 32 and 64 bits (and also less, e.g., 8 bits). Numbers up to roughly $2 \cdot 10^9$ fit in a 32-bit integer variable and $2 \cdot 10^{18}$ in a 64-bit one. Number overflow is one of the typical problems beginners upon which beginners stumble in contest. If you have three numbers $1 \leq a, b, c \leq 10^9$ the fit in an 32-bit integer. If you add them they will overflow and produce a negative number. You should use 64-bit precision in such a case. The same holds if you multiply two relative small numbers or add a big number of relatively smalls numbers: Overflow can occur.

In C and C++ use `long long` for 64-bit integer because `long` is often only 32-bit (depending on the compiler). In Java `long` is defined to be 64 bits. In Python you have infinite precision, which is sometimes an advantage.

Sometimes you are supposed to compute something that is really big, e.g., the 1000th Fibonacci number. Such numbers are so long that you are often asked to compute the result modulo some prime number. If this is the case do not forget to use the modulo operation on all intermediate results because otherwise you will get overflows and your final result will be wrong.

Here is a complete program that computes the nth Fibonacci number modulo 1073741723:

```
#include <iostream>
const int M = 1073741723;
int main() {
  int n; std::cin >> n;
  long long a = 0, b = 1, t;
  for(int i=1; i<=n; i++) {
    t = b;
    b = (a + b) %  M;
    a = t;
  }
  std::cout << a << "\n";
  return 0;
}
```

Whenever there might be a problem with overflows you should test your program with an input that uses big numbers. The sample inputs usually use only very small numbers.