Java Competitive Programming

“First, solve the problem.
Then, write the code.”
- John Johnson

By: Pedro Contipelli + Gabriel Gonzalez + Anirudh Rahul (Add your names here)

Contents

Unit 0: Getting Started

This guide is intended as a reference for students that are looking to learn the basics of Competitive Programming and don't know where to start. It will NOT contain everything you need to know, but it should at least point you in the right direction towards gaining the experience needed to do well in Competitions without having to do it the hard way (by failing a lot).

At the end of each unit, there will be links to problems that test knowledge of the topic discussed in that unit. All solutions have been tested in Java.

Create an account on Kattis Online Judge to practice submitting.

Hello World: Problem | Solution ← (These are links you can click)

Unit 1: Input / Output

Reading and Writing data (I/O) is one of the most essential things to know for competitive programming. The syntax may be different, but all libraries will work in generally the same way. Take in an input and return an output. It's as simple as that.

Most problems you'll encounter on online judges will input data from System.in (sometimes referred to as stdin), however some contests such as USACO will require the use of files.

We'll be using Java's Scanner library for the purposes of this guide, due to its user friendliness and ease of access. Scanner isn't very fast, but it should be sufficient for **most** contest problems. Advanced users might want to check out Kattio (Below).

Kattis has a custom-written library for fast and lightweight IO called Kattio. This is especially helpful for handling large amounts of input data on problems giving you a Time Limit Exceeded. To use it, create a new class on Eclipse and copy/paste the code provided into it. Make a separate class for your main program and write your solution there. The constructor and methods of a Kattio object are very similar to that of Scanner. Also make sure to submit BOTH your main program and Kattio class file or it will not compile on the judge's servers and you will get a Compile Error.

Scanner

Begin by feeding an input stream into Scanner's constructor. An input stream is simply

an object that inputs in data (i.e: System.in).

Example: Adding two integers from stdin using Scanner

```
import java.util.Scanner;
public class Guide {
public static void main(String[] args) {

Scanner scan = new Scanner(System.in);
int x = scan.nextInt();
int y = scan.nextInt();

System.out.println(x + y);

/*
* Although calling scan.close() isn't necessary for contest problems,
* it is recognized as good practice and will prevent memory leaks in
* larger code bases.
*/

scan.close();

        }
}
```

A lot of problem inputs will also give you some sort of a specification as to how much data they're going to be inputting. However, some problems will require you to keep scanning until the End of File or until a certain condition is reached.

Let's imagine our problem was to sum a list of numbers. Look at these three inputs.

Input #1	Input #2	Input #3
4 1 5 6 7	1 5 6 7 0	1 5 6 7

Input #1 first tells us how many numbers we have to sum (4) and then gives us the list.

Input #2 tells us to keep summing the list of numbers until a 0 is found.

Input #3 tells us to keep summing numbers until the end of the file is reached.

Example: Handling Input Type #1

```
int numbers = scan.nextInt();
int sum = 0;

for (int i = 0; i < numbers; i++)
        sum += scan.nextInt();

System.out.println(sum);
```

Example: Handling Input Type #2

```
int sum = 0;

while (true)
        {
        int add = scan.nextInt();

        if (add == 0)
                break;

        sum += add;
        }

System.out.println(sum);
```

Example: Handling Input Type #3

```
int sum = 0;

while (scan.hasNextInt())
        sum += scan.nextInt();

System.out.println(sum);
```

The last Scanner method we'll be covering is nextLine(). Reason being that this method exhibits some behavior that may seem counter-intuitive to many beginners.

Let's imagine a problem where we are given some lines of text, and have to concatenate an exclamation mark at the end of each line. Sounds easy enough, right?

Sample Input	Sample Output
3 Hello World Ouch My house is on fire	Hello World! Ouch! My house is on fire!

Common intuition tells us to write code looking similar to this.

Example: Beginner's Mistake

```
int lines = scan.nextInt();

for (int i = 0; i < lines; i++)
        {
        String line = scan.nextLine();

        System.out.println(line + "!");
        }
```

The above code will give you Wrong Answer.

Why is this wrong?

You can think of Scanner as a cursor highlighting text, like the one you see on the screen when you move your mouse. Now, let's think about what exactly is happening as the above code is running.

When you scan in the first integer that tells you how many lines will be in the problem, the cursor passes over the integer, but remains ON that line. There is nothing left to scan on that line, so the next time you call scan.nextLine(), you will simply get an empty String. And only then will the cursor move on to the next line.

How do we fix it?

Well, we just need to make sure the cursor moves on to the next line BEFORE we start running the loop. Code for doing this is on the next page.

Example: Beginner's Mistake (Fixed)

```
int lines = scan.nextInt();
scan.nextLine();

for (int i = 0; i < lines; i++)
        {
```

```
        String line = scan.nextLine();

        System.out.println(line + "!");
        }
```

You can check out the documentation for a complete list of all of its methods: Scanner

System

Most people are familiar with System.out, but not many know about System.err .

System.err is a print stream that comes in really handy during debugging.

Essentially, it provides you with a way to print output to the console in RED that will not be read by the judge. Using it, you will never have to worry about making the fatal mistake of submitting your program with leftover print statements again!

Example: System.err code

```
System.out.println("Hello");
System.err.println("This line of code has been executed.");
System.out.println("World!");
```

Console Output:

```
Hello
World!
This line of code has been executed.
```

Note: Due to a long standing bug in Eclipse, the relative ordering of the two input streams when used together is not consistent (As you can see in the example above).

Another really useful method to know is printf(), which stands for “Print Format”.
For a better understanding of how the formatting actually works: NumberFormat.

For now, you can just memorize how to use it for rounding numbers to a given amount of decimal digits. The format string you’ll want to use is “%.Xf” where X is the amount.

Example: Rounding double’s

```
System.out.printf("%.0f \n" , 45.250);
System.out.printf("%.2f \n" , 45.250);
System.out.printf("%.4f \n" , 45.250);

System.out.println();

System.out.printf("%.3f \n" , 5.0001);
System.out.printf("%.3f \n" , 5.0009);
```

Console Output:

```
45
45.25
45.2500

5.000
5.001
```

You may already know that println() will convert large double values to scientific notation. Another use of printf() is for getting the full decimal expansion.

Example: Decimal Expansion

```
System.out.println(Math.pow(10 , 50));
System.out.printf("%.2f \n" , Math.pow(10 , 50));

/*
* At UF-ACM 2016, one input set REQUIRED using printf()
* for printing large double values (as shown above).
*
* Programs using println() were judged as Wrong Answer
*/
```

Console Output:

```
1.0E50
100000000000000000000000000000000000000000000000000.00
```

Files

There's a couple of things you should know for handling File I/O:

- Simply instantiating a File object will not generate a File for you. The file must first be created and saved with the correct name. Capitalization IS important.

- It is possible to specify a file directory, but to make things simple, we'll be using the default directory: The Java Project Folder (NOT the /src folder). You must also make sure to throw an IOException in your main method when doing this or else it will not find the file.

Example: Reading two integers from input file and writing sum to output file

```
import java.io.File;
import java.util.Scanner;
import java.io.PrintWriter;
import java.io.IOException;

/*
* Some prefer to import and throw a FileNotFoundException instead.
* This is also suitable and more specific, but requires more typing.
*/

public class Guide {
public static void main(String[] args) throws IOException {

// Make sure to include the file extension (Such as .in or .txt)
File inputFile = new File("input.in");
File outputFile = new File("output.out");

Scanner scan = new Scanner(inputFile);
PrintWriter write = new PrintWriter(outputFile);

int x = scan.nextInt();
int y = scan.nextInt();

write.println(x + y);

scan.close();
write.close();

        }
}
```

Unit 1 Practice Problems

Stuck In A Time Loop: Problem | Solution
Speed Limit: Problem | Solution
Pizza Crust: Problem | Solution
The Cow-Signal: Problem | Solution

Unit 1 Challenge Problems

Hint: None of these problems can be solved under the time limit using Scanner

Splat: Problem | Solution
Compact Disc: Problem | Solution
Army Strength (Hard): Problem | Solution

Unit 2: Math

Computers don't do math like humans. This can be especially confusing to many beginners who are used to doing math in their math classes. Representing more abstract concepts (like infinity, for example) isn't really feasible for computers because they don't think the same way we do. Their language consists of 1's and 0's. And you will come to find that having a solid understanding of how things work behind the scenes (binary, logic gates, etc) will really help your brain think more algorithmically for competitive problem-solving.

Data Sets

Many problems will specify limits for their data sets. Do NOT ignore them. They usually hint at what the runtime of your algorithm should look like, and can be really helpful for debugging in the case of overflow errors.

For example, a problem might say: An integer n , $0 < n < 10^9$

The largest value an **int** variable can hold is $2^{31} - 1$.
The largest value a **long** variable can hold is $2^{63} - 1$.
Rough estimates: 10^9 fits in an integer. 10^{18} fits in a long.

You can also quickly access these bound values with predefined constants in the primitive wrapper classes: Integer.MAX_VALUE , Long.MIN_VALUE, etc

Number Theory

Certain questions may require you to find the **greatest common divisor** of 2 numbers. You could go with the naive approach which is a simple for-loop, but there is a more optimized algorithm that is also not much harder to code, known as the Euclidean algorithm. While you could code this algorithm easily it is also built into the BigInteger class you can read more about the method here:
https://www.tutorialspoint.com/java/math/biginteger_gcd.html

Example: Finding greatest common divisor recursively

```
public static int gcd(int a , int b) {

	if (a == 0)
		return b;

	return gcd(b % a , a);
}
```

This algorithm may not make sense at first but after you think about it for a while it makes much more sense. It builds upon the fact that if **a** and **b** are both divisible by **c** then **a-b** is also divisible by **c**.
Ex.
900 and 880 differ by 20 so their gcd would have to be 20 a factor of 20.
The algorithm for the **least common multiple of n numbers** is also relatively simple it is simply the product of the **n** numbers divided by their gcd.

Example: Finding LCM of 2 numbers

```
public static int lcm(int a, int b){
	return a*b/gcd(a,b);
}
```

If you want to find the gcd of a large list of numbers just keeps gcding adjacent numbers ex.
gcd(a,b,c)=gcd(gcd(a,b),c)

```
public static int ListGcd(List<Integer> list){
	int gcdOfList=list.get(0);
	for(int n:list)
		gcdOfList=gcd(gcdOfList,n);
	return gcdOfList;
}
```

Searches

Searching is a key part of Competitive programming. Searching problems usually involve graphs that you must “recreate” and search through, they will most likely ask you for a path from node A to node B. Graph theory can come handy but your best tools are searching algorithms.

Breadth-first search: It checks all nodes adjacent to node A, then the nodes adjacent to those nodes until it reaches node B. If all reachable nodes are checked and node B is not found there is no path.

Depth-first search: This search moves in a certain direction until it hits a boundary, in which case it either changes direction or backtracks. If it backtracks to where it started, there is no path.

Both searches have the same time complexity so you can use whichever one you like better.

BFS example