How to GCPC

The GCPC Jury

January 28, 2026

Contents

This document is intended to provide some guidance for teams participating in the Winter Contest or the GCPC for the first time, but may also be a useful reference for more experienced contestants. It includes tips and tricks for all parts of the contest, from reading the problem set to designing an algorithm to coding that algorithm to making a correct submission.

1 Before submitting

1.1 Picking a problem to solve

When you first open up the problem set at the start of the contest, there will be a large number of problems to choose from, so where do you even begin solving all of those? The first thing to note is that the problem set covers a wide range of topics and difficulties, so you should make sure to read all the problems between the members of your team and then work on the problems that look the most fun and approachable to you. In the Winter Contest, the easiest problems are marked on the title page of your problem set, and those should be a great place to start!

You should also keep an eye on the scoreboard to see which problems other teams are solving (or not solving), as problems can often be easier or harder than they appear at first sight. Especially as a beginner team you should make sure not to sink a lot of time into a really hard problem.

1.2 Solving problems and designing algorithms

This is of course heavily dependent on the problem at hand, but here are some very general ideas and techniques that are widely applicable:

- Find an abstract formulation of the problem. Does it relate to something familiar?
- Sorting, binary search, divide and conquer, greedy, dynamic programming.
- Work out some small examples on paper. Try to spot any patterns.
- Use the simplest algorithm that fits the constraints. If you try to be super smart you might lose a lot of time.

1.3 Estimating the running time

Before implementing your algorithm, it's important to check that it has the right time complexity. Many problems have a relatively straightforward solution that is too slow for the given bounds, which means that your task is to come up with a better algorithm! To help you gauge whether your approach is good enough, here's a rough estimate of the maximal input sizes that an algorithm with given time complexity will be able to handle.

$O(n!)$	$n \le 10$	$O(n \log^2(n))$	$n \le 10^5$
$O(2^n)$	$n \le 20$	$O(n \log(n))$	$n \le 10^6$
$O(n^3)$	$n \le 500$	$O(n)$	$n \le 10^8$
$O(n^2 \log(n))$	$n \le 1\,000$	$O(\sqrt{n})$	$n \le 10^{15}$
$O(n^2)$	$n \le 5\,000$	$O(\log(n))$	$n \le 10^{18}$
$O(n\sqrt{n})$	$n \le 10^5$		

In general, **it is much more important to have the right time complexity than it is to optimize for runtime constants**. The input bounds and timelimit are usually quite lenient, and an efficient implementation of the intended algorithm will typically only use fractions of the allotted running time. Conversely, algorithms with worse time complexity will generally time out regardless of the amount of constant optimization that is put into their implementation.

It is also important to note that time complexity always refers to the *worst-case* time complexity, and heuristic approaches are unlikely to pass all the secret test data, even if they are fast in the average case.

This being said, not all operations are created equal. This is likely not a problem, but considering it may help if your algorithm has the right time complexity but is still too slow. For instance, if your code is $O(n)$, but also needs to store that amount of data in a dictionary, then something like $n \leq 10^6$ is more realistic than the $n \leq 10^8$ from the above table. Potentially slow operations include:

- Usage of hashing data structures, such as `unordered_map`, `HashSet`, `dict`.
- I/O operations; see Section 4.2 for speedups, but it will always be relatively slow.
- Memory access; the smaller your memory usage, the less caching there needs to be.
- Floating point functions such as `sqrt`, `log`, `exp`, `pow` and trigonometric functions.

Other factors that come into play are the choice of programming language (C++ is generally faster than Java and Python), and the time limit of the specific problem.

1.4 Implementation

Use the language that you or your team is most comfortable with. Each submission must read from the standard input and write to the standard output. Here are sample implementations of a simple A+B program in C++, Java and Python.

- C++:

```
#include <iostream>
using namespace std;

int main() {
  int a, b;
  cin >> a >> b;
  cout << a+b << '\n';
}
```

- Java:

```
import java.util.*;

public class A {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    System.out.println(a+b);
  }
}
```

- Python:

  ```
  a, b = map(int, input().split())
  print(a+b)
  ```

All of the contest's languages already provide many different algorithms and data structures as part of their standard library, such as sorting, binary search trees and hash tables. Use those instead of implementing your own.

While programming contests are ultimately about speed, you should still try to keep your code as simple and clean as possible, as that makes it far easier to debug if things go wrong.

It is generally a good idea to get at least a rough overview of how the entire program is going to look before diving into the implementation right away. Ideally, your code is written all in one go and you don't need to spend any time refactoring.

1.5 Running and testing your code

Always test your code on *all* the provided sample data (which you can download from the judging system). The following assumes you are using a command line environment. Either open one directly or use the one that's included in your IDE or text editor. To build/run your program, use the following commands:

	Build	Run
C++	`g++ -std=c++20 -g A.cpp -o A`	`./A`
Java	`javac A.java`	`java A`
Python	-	`python3 A.py`

To test your code on the provided sample data, download the zip archive from the website, extract it to a directory `samples`, then use the following (this is for C++, replace `./A` with the appropriate run command for your chosen language, see above).

```
./A < samples/A/1.in | diff -s - samples/A/1.ans
```

Do this for all samples. Depending on the problem, the output may not be unique, in which case you need to inspect the output of the `diff` carefully. If the problem is interactive, follow the instructions in the provided testing tool.

You may then want to test your code on some handcrafted cases, especially if the input format makes it easy to do so. Sometimes it can also be a good idea to write a simple script to generate additional tests, but of course this takes up some of your team's valuable screen time.

2 After submitting

If all went well, your submission was accepted and you can move on to the next problem while waiting for your well-deserved balloon, congratulations! If not, here are a few checklists to help you with your trouble shooting, grouped by the type of verdict your submission received. Note that the boundaries between these categories are not sharp, as some of these mistakes may manifest in different errors. Most notably, memory access violations in C++ may sometimes cause your program to read garbage values without crashing. Your submission may also receive multiple different negative verdicts, but you are only shown the first one to occur.

2.1 General advice

- Click on your submission to see its results on the sample cases.
- Did you submit on the right problem? Did you submit the right file?
- Think about possible corner cases (e.g. very small ($n = 0, n = 1$) or very large inputs) and test your submission on those (Section 1.5).
- Print your code, let a teammate continue with a different problem.
- Do some debugging, either using some debugging tool or some simple `printf` statements. Be aware that this eats into your precious screen time though!
- Try to explain your code line by line to a teammate.
- Consider shelving the problem for later or abandoning it altogether.

2.2 Wrong Answer

- Check the given sample cases again (Section 1.5).
- Try to find possible corner cases you have not yet considered or tested.
- Read the problem statement again. Did you understand the problem correctly? Are you making any assumptions that are not justified by the problem statement?
- In C++/Java, check for integer overflow (Section 4.1).
- In C++/Java, **`float`** is too imprecise, use **`double`** instead (Section 4.1).
- Outputting floating point values: print with enough precision (Section 4.1).
- In C++/Java, `%` may return negative numbers, e.g. `(-5) % 3 == -2`.
- Don't use the `pow` function in C++/Java/Python, as it returns floating point numbers.
- Avoid doing comparisons (<,<=,==,...) on floating point numbers, as these are prone to fail due to rounding errors.
- Are you doing floating point calculations that may result in NaN?

2.3 Time Limit Exceeded

- Make sure that your code has the right time complexity (Section 1.3).
- Also check the time incurred by library functions. For instance, random access operations on certain data structures may use a linear amount of time. Consult the documentation.
- Check for infinite loops or endless recursion, especially on corner cases.
- Are you outputting large amounts of text to `stderr`?
- Function arguments in C++ are passed by value (copy). Use `&` to pass by reference.
- Interactive Problem: do you flush after your output?

2.4 Run Time Error

- Check for divisions by `0` or other math errors such as `sqrts` of negative numbers.
- In Python, importing external libraries such as NumPy causes a run time error.
- Do you have any assertions in your code that may fail?
- Are you calling library functions with invalid input? Check the documentation.
- Does your program try to allocate more than the memory limit (2GB)?
- Check for out of bounds access on arrays/vectors.
- Deep recursion may cause a stack overflow, especially in Java and Python.

2.5 Compiler Error

- Click on your submission to check the error message.

3 Problem Types

GCPC and Winter Contest not only offer a diverse set of problems, but these problems may also work differently when judging your submission. Generally speaking, there are three distinct “problem types”: Pass-Fail, Interactive, and Multi-Pass. Not all problem types are necessarily present in every contest.

3.1 Pass-Fail Problems

The Pass-Fail Problem type is by far the most common in GCPC and Winter Contest. For these problems, your program is given a *single (static) input* and should output the solution in the given time and memory limit.

Depending on the problem, there may be multiple solutions. In this case, the output section of the problem should specify which solutions are accepted. Most commonly, any valid solution may be accepted, but for some problems, your program should output the smallest or largest possible solution instead.

3.2 Interactive Problems

The second problem type are Interactive Problems. Contrary to the Pass-Fail Problems, your program is not provided with a static input. Instead, your submission will run against an *interactor.*

The interactor will read your program's output, evaluate it, and write the next input to your program. Therefore, the interactor can respond to choices made by your program. The given time and memory limits apply to the complete run of your program. For example, if your program does not properly implement the interaction protocol and waits for the interactor, which in turn waits for the output of your program, your submission will obtain the error "time limit exceeded".

Therefore, special care must be taken to properly *flush* your output so that it will be sent to the interactor. For example, you can use `fflush(stdout)` in C++, `System.out.flush()` in Java, `sys.stdout.flush()` in Python, and `hFlush stdout` in Haskell.

There should be a testing tool provided for Interactive Problems. You can run the testing tool locally to check your submission for common errors. Note that this tool only performs basic checks on your submission, e.g., that your program conforms to the correct protocol used by the interactor. The fact that your submission passes the testing tool does not guarantee it will pass the judge. For example, there may be additional secret test cases on which your submission fails.

3.3 Multi-Pass Problems

In Multi-Pass Problems, your program is executed multiple times for each test case. Each separate invocation for the same test case is called a "pass". For each pass after the initial one, the input provided to your program may depend on the output generated in previous passes. Depending on the problem and output validator, there may either be a predefined number of passes, or a variable number of passes for a given test case. The problem statement describes the input format and constraints for each pass. In any case, time and memory limit apply to each pass individually.

There should be a testing tool provided for Multi-Pass Problems. You can run the testing tool locally to check your submission for common errors. Note that this tool only performs basic checks on your submission, e.g., run your program multiple times using output from previous passes. The fact that your submission passes the testing tool does not guarantee it will pass the judge. For example, there may be additional secret test cases on which your submission fails.

4 Miscellaneous Tips and Tricks

4.1 Builtin Data Types

- In C++/Java, `int` can only represent numbers up to $2^{31} - 1 \approx 2 \cdot 10^9$. For larger integers, use `long long` in C++ or `long` in Java, which both range up to $2^{63} - 1 \approx 9 \cdot 10^{18}$.
- Never use `float` in C++/Java, prefer `double` instead. Note that Python's `float` is equivalent to the latter. Avoid using floating point numbers as much as possible, as they are susceptible to rounding errors (e.g. `3*0.3` does not equal `0.9` in floating point arithmetic).

In C++, floating point numbers are only printed to 5 digits of precision by default, which is often not enough. To print more digits, use

```
cout << fixed << setprecision(20) << sqrt(2) << '\n';
```

It is generally a good idea to always print more digits than you think are required.

4.2 Fast I/O

If your submission needs to process large amounts of input, you can use the following tricks to speed it up. However, note that the timelimits are generally set so that none of these optimizations should be required. In particular, if your algorithm has the wrong time complexity (see Section 1.3) your submission will very likely be too slow even with heavily optimized I/O.

- In C++, add `cin.tie(0)->sync_with_stdio(0);` at the start of `main`. Note that if you do this you can no longer mix `cin` and `scanf`. Use `'\n'` instead of `endl`.
- In Java, `Scanner` is slow. Consider using `BufferedReader` instead.
- In Python, PyPy generally optimizes I/O quite well. If you need more speed, you can use `io.BytesIO`.

4.3 Compiler Options for C++

In C++, you can use the following additional flags for `g++` which try (but do not guarantee!) to catch undefined behaviour and segmentation faults at run time:

```
  -ftrapv -fsanitize=address,undefined
```

The following flags check for various common mistakes at compile time:

```
  -Wall -Wextra
```

Check the team information sheet for the compiler flags used by the judging system.