

AACPP WiSe

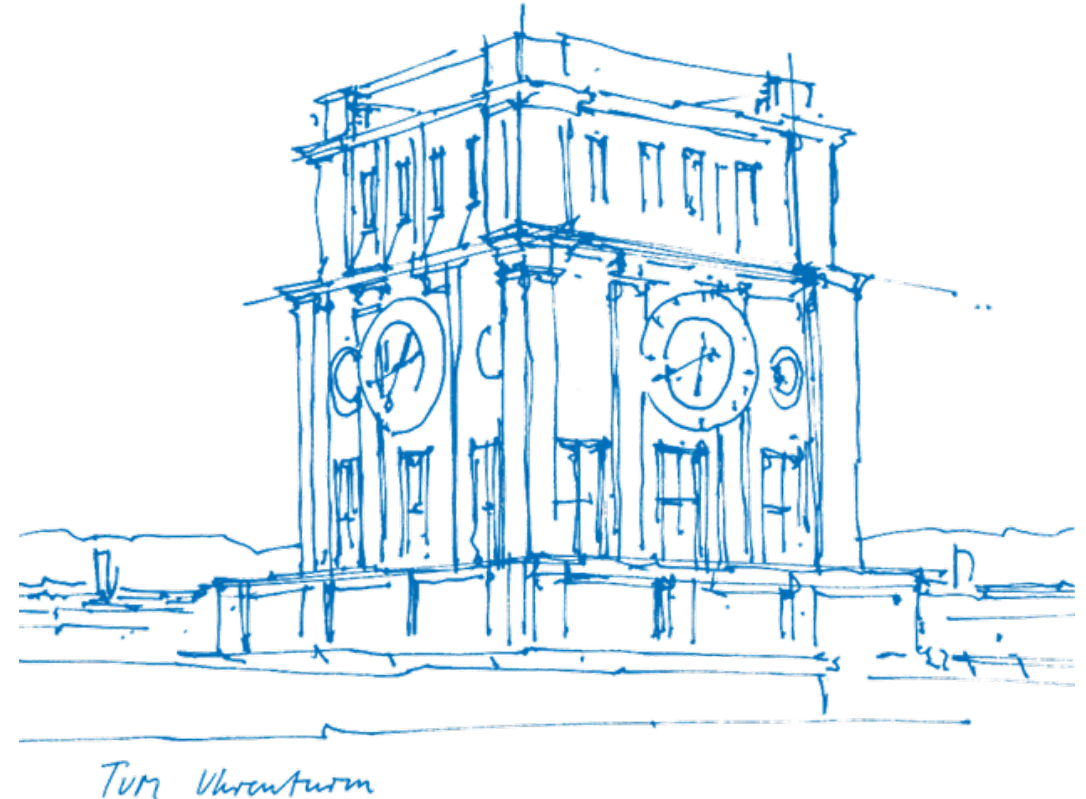
2026/27

Intro

Mateusz Gienieczko, Julia Spindler

School of Computation, Information and Technology
Technical University of Munich

2026.10.15



Structure of the course



Weekly meetings here, Thursdays, 12:00.

We will focus on explaining solutions to tasks.

Then a short introduction of the new topic and tasks.

7 **tasks**, each for 10 + 10 points.

Task published biweekly on Thursdays, two weeks to solve.

Expect 71 points to pass.

Regulations



Course rules and technical spec on course page:

db.in.tum.de/teaching/ws2627/aacpp

Problem statements



Always read the problem statement carefully.

The **Main section** contains the actual problem to be solved.

Input contains formal input description.

Output contains formal output specification.

Limits set the size parameters of the task.

You upload a single file in `.rs`, `.cpp`, or `.c`.

It gets compiled and ran on **multiple test cases**.

Your program must read from **standard input** and write to **standard output**.

Your output is checked for correctness.

The environment is constraint wrt. capabilities, **time**, and **memory**.

Translate from author's imagination into a more formal version.

All test cases conform to the formal input description in the test.

Always assume the input is correct (in the sense described in the Input section).

Read from standard input.

Your output must conform *strictly* to the format.
The only allowed difference is trailing whitespace.

Each task contains at least one example of valid input and a correct answer.

They're unlikely to be interesting, they're a smoke test.

Limits are very important.

They hint at the complexity of the correct solution.

Ok, let's submit



The system: judge.db.cit.tum.de

All submissions happen through the system.

All submissions are automatically judged.

Tests are grouped.

E.g. 1a, 1b, 1c are all in the same group.

You get points for a group if *all* tests in the group are OK.

Coding solutions for competitive programming is different than actual coding.

Coding solutions for competitive programming is different than actual coding.

BUT

it **is** very educational!

Problem solving



We have a giant toolbox at our disposal and a problem to solve.

Usually not as well defined, but translating requirements to something sensible is similar.

The main difference is there is no judging system.

Problem solving



We have a giant toolbox at our disposal and a problem to solve.

Usually not as well defined, but translating requirements to something sensible is similar.

The main difference is there is no judging system.

No one will ask you to prove complexity at a Real Job™...

BUT

they will ask “will this work if we have a million entries”.

Problem solving



We have a giant toolbox at our disposal and a problem to solve.

Usually not as well defined, but translating requirements to something sensible is similar.

The main difference is there is no judging system.

No one will ask you to prove complexity at a Real Job™...

BUT

they will ask “will this work if we have a million entries”.

Try not to solve tasks by a thousand resubmissions, for your own sake.

- Make simplifying assumptions
- Start from suboptimal but “obviously correct” solutions (brute force)
- Generalise from there

Oh no it doesn't work



If it doesn't work on example or your manual tests – good!

What if we don't have a counterexample?

Proving correctness



Sometimes solution doesn't work fundamentally.

Sometimes it's a coding bug.

Hard to find a bug if you're not sure that your solution is supposed to work.

Usual approach – vibes well means it works



Quite useful for competitive programming where time is limited.

Dangerous, greedy solutions often vibe well but might not work.

At least try to prove it to yourself



Often you'll talk yourself into a counterexample.

... or maybe it actually works conceptually and then you need to debug.

Generating tests



Random tests are usually poor quality.

Generating tests



Random tests are usually poor quality.

BUT

We can generate *a lot* of them.

Generating tests



Random tests are usually poor quality.

BUT

We can generate *a lot* of them.

Professionals call this “fuzzing”.

Output?



Sometimes it's possible to generate a test for a given output, but not always.
Brute-force solutions to compare against.

USE A DEBUGGER.

If you're already using one, good, keep doing so.

USE A DEBUGGER.

If you're already using one, good, keep doing so.

If not and you like vim you will also like gdb.

USE A DEBUGGER.

If you're already using one, good, keep doing so.

If not and you like vim you will also like gdb.

Otherwise, use an IDE. CLion is really good.

Printf debugging



Slapping some print statements to print variables and overall flow of the program.

Actually pretty effective in competitive programming.

Printf debugging



Slapping some print statements to print variables and overall flow of the program.

Actually pretty effective in competitive programming.

Professionals call this “structured logging”.

International Collegiate Programming Contest

There's also GCPC.

There is a group on TUM doing ICPC. They sometimes have posters.

[Telegram link](#)

Other contests



Codeforces

LeetCode

TopCoder

SPOJ

CodeChef

CodinGame

CodeWars

CodeCombat

Introduction to Algorithms (4th ed.), Cormen et al. (The Algorithms Bible)

Algorithms and Data Structures book by the CS department of Virginia Tech

Competitive Programming Resources by Kunal Kushwaha

Programming Challenges: The Programming Contest Training Manual, Skiena et al.

Competitive Programming book(s) by S. Halim, F. Halim and S. Effendy

Competitive Programming Handbook and others by Antti Laaksonen

Course resources

Official Zulip

Evaluation system

Course website

