**DEBUGGING TIPS DROM PEOPLE**

<https://www.quora.com/How-do-you-debug-your-code-quickly-in-programming-contest-environments>

[](https://www.quora.com/profile/Nikhil-Garg)

[Nikhil Garg](https://www.quora.com/profile/Nikhil-Garg), ACM-ICPC World Finalist (2011, 2012)

[Updated Oct 1, 2012](https://www.quora.com/How-do-you-debug-your-code-quickly-in-programming-contest-environments/answer/Nikhil-Garg) · Author has **268** answers and **5m** answer views

Programs in programming competitions are small and don't involve difficult real world issues like race conditions, device failures and network errors. That is why debugger is mostly an overkill here. The only thing that truly matters is saving some time. Different programming competitions impose different restrictions so a universal answer is probably not possible, here is my advice for ACM-ICPC and similar formatted competitions :

1. Write clean code. Use meaningful and yet not very long variable names. Use white spaces generously. Maintain the indentation.
2. Structure your code cleanly - distrubute your logic into small functions. If one of your functions is > 50 lines, there's a good chance you're not doing it the right way.
3. As soon as you get an error, read your code quickly. If you've followed steps 1 and 2 carefully and know what your program is supposed to do, you probably will be able to find your bug at this stage. With experience you'd learn to catch most common bugs at this stage itself. In particular, keep your eyes wide open for integer overflow errors.
4. Most online judges give you an indication if you're getting wrong answers or if your program is exceeding the given time limit or if it is simply crashing. Keep the error type in your mind as you move forward with debugging.
5. Create some sanity-check trivial and boundary test cases to check if your program gives correct answers on those. If not, run through your code in the same order it will be executed on the simplest case on which it doesn't work.
6. Start implanting print statements following a binary search type order. Find the first place where program goes wrong.
7. Specifically to quicken the above process, people often keep handy functions in their arsenal which can print variables quickly. e.g look at following Java function :  
   1. **void** debug(Object...ob)
   2. {
   3. System.**out**.println(Arrays.deepToString(ob));
   4. }

You can call debug() with any number of arguments of any type (all basic type and all containers) and they'd be printed in a neat and easy to read format. Writing a similar function in C++ isn't that easy but I've written one here : [What are some macros that are used in C++ programming contests?](https://www.quora.com/C++-programming-language/What-are-the-basic-macros-that-are-used-in-programming-contests/)

1. If all this is not useful so far, you'd need to slow down and figure out if its a logical bug or an implementation bug. At this stage, I usually try to prove my solution's correctness. If it is indeed a logical bug, get back to correcting it. If not, move forward.
2. If its not a logical bug - try to think of extreme cases like stack overflow, memory overflow or an infinite loop coming up on specific test cases.
3. As last resort, generate some random test data, write a brute force solution and diff outputs of your program with that of the brute force checker. There is a good chance you'd find an error here. Pick up the simplest case which went wrong and go back to step 5.
4. If you couldn't find a bug even on random test data, and see no hope of being able to debug your program, you could enter a very costly debugging mode. Implant conditions like this at various places (in order of binary search of course):  
   1. **if** program **is** correct till **this** stage
   2. **then** go **into** infinite loop
   3. **else**
   4. crash the program

And submit the program on online judge several times (while changing the location of these lines). Looking at the exit status of the program, it is possible to trace the mistake on judge data. Its downside of course is that it takes a lot of time and costs several submissions but in ICPC like competitions, a single correct submission supersedes any number of wrong submissions.

Having said all this, in time, I've adopted the following strategy for myself which tries to get the solution right in first go itself:

1. Prove correctness of the solution before writing a single letter.
2. Write very clean and neat code, carefully structured.
3. Write some asserts while writing the first draft of the code itself.
4. In case there is a bug, read all parts of code several times and implant debug() calls at relevant places.

This approach takes more time in getting to the first execution of the program but on an average, saves tremendously on debugging time.

[](https://www.quora.com/profile/Pradeep-George-Mathias)

[Pradeep George Mathias](https://www.quora.com/profile/Pradeep-George-Mathias), IOI-2008, ICPC WF 2011,12

[Answered Oct 1, 2012](https://www.quora.com/How-do-you-debug-your-code-quickly-in-programming-contest-environments/answer/Pradeep-George-Mathias) · Upvoted by [Nikhil Garg](https://www.quora.com/profile/Nikhil-Garg), ACM-ICPC World Finalist (2011, 2012) · Author has **103** answers and **1.2m** answer views

Being from the same ICPC Team as [Nikhil Garg](https://www.quora.com/profile/Nikhil-Garg), most of my answer would match his. I am just presenting my own take on the entire process.

1. Write clean code. Use meaningful and yet not very long variable names. Use white spaces generously. Maintain the indentation.
2. Structure your code cleanly - distrubute your logic into small functions. If one of your functions is > 50 lines, there's a good chance you're not doing it the right way.

These 2 steps are definitely the *most important*.  
Essentially,

* *clean code =>***easier readability**. You can skim over the code fairly generously and quickly.
* *small functions* => ask yourself: *Is this block of code doing*exactly*what it needs to be doing?* By this, you can **eliminate entire blocks of code**from being buggy.

After these two steps, you can quickly identify if the program flow is pretty much ***exactly as your logic dictates***. **Confirm this.**  
  
At this stage, I'd advise you skip down to Nikhil's point 8. (or his 2nd approach point 1.)

8. If all this is not useful so far, you'd need to slow down and figure out if its a logical bug or an implementation bug. At this stage, I usually try to prove my solution's correctness. If it is indeed a logical bug, get back to correcting it. If not, move forward.

1. Prove correctness of the solution before writing a single letter.

**Prove your logic is sound!**At this point, you must examine

* **Boundary Cases:** Things that are in some sense "special cases" or which would go beyond the scope of your "proof of correctness"
* **Assumptions** about your proof: See if there is something you're assuming that is intuitive, but not stated. (For eg. A *lot* of geometry problems have flawed assumptions arising out of visualizing the instances. ***Question your assumptions***). Boundary cases often arise out of misplaced assumptions in fact.

Finally, *most* of my debugging phase consists of iterating between these 2 steps: the validation of implementation, and the proof of correctness (not necessarily in this order).  
  
Nikhil's other points detail checking for implementation bugs. The use of prints can often identify both logical as well as implementation bugs.