PROBLEM SOLVING TECHNIQUES (For Beginners particularly)

(Final Class from BAPS-BUBT Programming Training Camp 2016, Tips and Q/A session)
(Lecturer: Hasnain Heickal Jami sir)

When we focus on Problem Solving, different kind of issues arise. The following is a summarization of the entire class and skims through the critical problems, for both personal problem solving and onsite team contests. I have subdivided the main issues into 3 parts. The rest can be solved through practice and experience.

Part 1: Interpreting Problems

Throughout the whole process of solving a given problem, we all have been to through a situation where everything seems to be right, yet we get a verdict of "Wrong Answer". For most cases the situations are a result of overlooking constraints, not thinking about critical cases, or just a simple wrong approach. The strategies to face such problems are:

1. During onsite contests, when you get a printed copy of the problem set, highlight all seemingly important points of the description of the problem. Highlight the input and output constraints particularly.
[As for normal contests, or problem solving, write down all the constraints separately in a notebook.]

2. List and check through all possible easy strategies first. Set aside anything that doesn't help with the approach.
[All seemingly hard problems do not always necessarily need an advanced solution.]

3. Brainstorming - Go through the constraints again and again to find out loopholes in the question to be exploited, or create a critical test case.

4. There might be *[will be!]* problems that discuss a daily life scenario. One of the most critical mistakes made by contestants is assuming that the problem will also follow real world rules. Never relate the example of the question with the real world object. Always focus on the question.

5. Never assume any unmentioned conditions. There are some problems -especially when it comes to graphs - where it might not be mentioned whether a graph is directed or not. In that case we must rely on the test cases to check which approach is supported by the sample outputs.

6. Another problem with graphs is not mentioning whether a cycle is present or not. For that we must assume there is, as a cycled graph is a superset of non-cycled graphs, so our solution will work. And apart from this, another mistake is to assume that the graph is a tree. Undirected graph can be a tree, but we must assume that the graph is a directed acyclic graph.
[Simple assumption also needs a lot of experience, eh?]

7. And last but the not the least, whenever a seemingly solid approach is ready, prepare the simulation & work through to solve sample cases on notebook. Prepare the code, and then test all critical test cases. It might seem to be a long procedure, but spending 10 minutes on preparing sample scenarios is definitely better than getting a "WA", earning a 20 minute time penalty and **then** start to work and find some critical cases.
[A lot of practice is required to muster the patience. Don't give up! ☺]

Part 2: Personal/Team Problem Solving Tips:

1. Visualizing a problem is the essence of problem solving. Avoid taking on pressure while solving. Try to stay relaxed and subdivide the problem into sub problems where the respective sub solutions are easy to visualize.
*[**Example**: Find the area of the zone where two given 3D cubes are touching.]*
*[**Approach**: Reduce the 3D cubes to 2 squares on 2D and check if they cross. If required, further reduce the 2D squares into 2 1D lines and check if the ending point of one line exceeds the starting point of the other. Visualize.]*

2. At first, if the constraints are confusing, try approaching the problem without any constraints. Once you are able to reach a solution, try introducing the constraints one by one. This will give a clear idea about how each constraint is affecting the solution and is essential for cooking up critical case.

3. Once the prepared solution passes the given constraints, try imposing new constraints to reduce options further. This will result in simpler, more elegant algorithms that will work with most difficult constraints.
*[**A simple example**: Problem asks you to color a graph with 3 colors. To take it a step further, try to complete the graph by bicoloring, and then just apply the color to all remaining nodes. ☺]*

4. Apply "**Brute Force**": Try dividing the main problem into small set of problems that can be brute forced. The simple loops are the best options for crosschecking the answers of self made test cases, confirming solutions to the main problem and can also be used for debugging within certain ranges.
*[Very useful recovery option for **team contestants!**]*

5. Never ask for code for fellow programmers or seniors. Try your level best, and if the solution still seems distant, ask for hints or clues for certain errors. Never ask anyone else to fix your code for you.

[NEVER LOOK UP SOLUTIONS FOR A PROBLEM YOU CAN NOT THINK UP A SOLUTION FOR ON THE 1st TRY!]

Part 3: TEAM CONTESTS!

1. HAVE FAITH ON YOUR TEAM MATE!

2. The main target is to use the three members to their max efficiency. One effective approach: Use 1 member to read through the problem set while the other 2 starts on one problem. This will help to filter out the seemingly impossible to solve ones and save time by calculating the approximate time required for the mid level problems.

[If there is enough time left afterwards, your team can try on the hard ones.]

3. Understand a problem clearly using tips from Part 1, and then explain the problem to your team mate/mates. This will save the time that would have been wasted if your teammate had to read the problem himself/herself. But be careful, while explaining, never impose your own ideas on him/her. First give them the chance to understand and formulate their own approach, and then compare which idea is a seemingly better solution.

[This often results in different angles coming from each team mate and very helpful for creating critical test cases.]

4. While you are coding, discuss with your team mate to let him/her know what is it you are trying, and ask for their points of view. Make sure at least one of your team mates is preparing strong and critical test cases for your solution.

5. Contribution must be distributed equally. Never try to take on all the pressure yourself, particularly for typing. When it is a repetitive process, our mind gradually loses focus, and very silly mistakes are made. Try member switching, and impose rules like any single member will not type 2 consecutive problems.

6. Don't get stuck on a "WA". Work parallel to reduce the recovery time. Once all the correct methods for solving a problem has been used, it will be easy to locate any certain error and fix it.

[Remember, the team with a shorter recovery time will have a better chance to lead! ☺]

7. Never block the team PC if you score a WA! Working on the same solution while blocking the PC is just wasting valuable time. Take printouts of your code, or simply use pastebin to move your solution to a different PC, and work on locating the error in your code. Never get stuck on solving one single problem. Try to switch to a different problem if you hit the dead end with one. This will help in the long run, as more temporary solutions will be available, and test cases will be ready, so you can work on and submit multiple solutions at the same time.

[Never take on more than 3 problems at a time, this will tilt the balance, and solutions will be vague and messy.]

8. **THE MOST IMPORTANT PROBLEM, TYPING SPEED:** Standard ACM ICPC world finalists have typing speed of above 80 words/min. Practice touch typing. Memorizing the keyboard through muscle memory and learning keyboard shortcuts helps a lot when the main challenge is to finish the solution on time and submit. Most people complain about team mates typing speed, and hence the team fall off balance. Try giving the simple problems to the slower typing team mate, while you watch from the side for mistakes and prepare test cases. This will result in you being able to type and keep your cool in time of dire need.

Just an attempt to share the knowledge with fellow programmers. Please overlook any typing errors and let me know of any mistakes.* HaPpY CoDinG *!!* ☺ *