How to Dissect a Topcoder Problem Statement

By **antimatter** — *topcoder member*
Discuss this article in the forums

How many times has this happened to you: you register for the SRM, go into your assigned room when the system tells you to, and when the match starts, you open the 250… and find it incomprehensible.

Maybe it's never happened to you. You may be lucky, or you may already be extremely skilled. However, many experienced competitors (yes, reds too) might end up just staring at a problem for a long time. This is a pretty serious issue. How can you solve the problem if you have no idea what it's asking you to do?

Fortunately, topcoder problem statements are formatted in a very specific way.

Knowing your way around the various pieces will go a long way towards helping you understand what the problem is saying.

The Parts of a Problem Statement
Let's look at the composition of a typical topcoder problem statement. First off is the introduction. Usually, a problem will be led off with a high-level description of a situation. This description may tie into real-life ideas and topics or it may just be a completely fictional story, serving only as some sort of context. For many problems the back-story itself is not particularly important in understanding the actual problem at hand.

Next comes the definition section. It gives you the skeleton of the solution you need to write: class name, method name, arguments, and return type, followed by the complete method signature. At minimum, you will need to declare a class with the given name, containing a method which conforms to the given method signature. The syntax given will always be correct for your chosen language.

Sometimes notes follow the method definition. They tend to be important reminders of things that you should pay attention to but might have missed, or they can also be things that are helpful background knowledge that you might not know beforehand. If the notes section appears, you should make sure to read it – usually the information contained within is extremely important.

The constraints section is arguably the most important. It lists specific constraints on the input variables. This lets you know crucial details such as how much memory to allocate or how efficient your algorithm will have to be.

Finally, a set of examples is provided. These give sample inputs against which you can test your program. The given parameters will be in the correct order, followed by the expected return value and, optionally, an explanation of the test case.

- **Introduction**
  The problem statement usually begins by motivating the problem. It gives a situation or context for the problem, before diving into the gory details. This is usually irrelevant to solving the problem, so ignore it if necessary. In some cases, the motivation can cause serious ambiguities if it is treated as binding – see MatchMaking (SRM 203 Div I Easy / Div II Medium). Also note that for some simple problems, the initial context may be left out.

  The ordering of the rest of this section varies greatly from problem to problem, based on the writing style of the problem author.

  There will be a description of what you need to do, in high-level terms. Take, for example, UserName (SRM 203, Div 2 easy). What the problem is asking for you to do is to find the first variant of a given username that is not already taken. Note that the problem has not yet said anything about variable names or types, or input formats.

  There will also be a low-level description of the input. At the bare minimum, the types and variable names of the inputs will be given to you, as well as what they correspond to and what they mean. Sometimes much more information about input formats will be given; this typically occurs in more complicated problems.

  Sometimes, even more detailed background information needs to be provided. That is also typically given here, or sometimes in the Notes section.

- **The Definition**
  This is a very barebones description of what topcoder wants you to submit. It gives the class name, the method name to create inside that class, the parameters it should take, the return value, and a method signature. As mentioned before, the basic form of a submitted solution is to create a class containing a method with the required signature. Make sure that the class is declared public if not using C++, and make sure to declare the method public also.

- **Notes and Constraints**
  Notes don’t always appear. If they do, READ THEM! Typically they will highlight issues that may have come up during testing, or they may provide background information that you may not have known beforehand. The constraints section gives a list of constraints on the input variables. These include constraints on

sizes of strings and arrays, or allowed characters, or values of numbers. These will be checked automatically, so there is no need to worry about writing code to check for these cases.

  Be careful of the constraints. Sometimes they may rule out certain algorithms, or make it possible for simpler but less efficient algorithms to run in time. There can be a very big difference between an input of 50 numbers and an input of 5, both in terms of solutions that will end up passing, and in terms of ease of coding.

- **Examples**
  These are a list of sample test cases to test your program against. It gives the inputs (in the correct order) and then the expected return value, and sometimes an annotation below, to explain the case further if necessary.

  It goes without saying that you should test your code against all of the examples, at the very least. There may be tricky cases, large cases, or corner cases that you have not considered when writing the solution; fixing issues before you submit is infinitely preferable to having your solution challenged or having it fail during system testing.

  The examples are not always comprehensive! Be aware of this. For some problems, passing the examples is almost the same as passing every test case.

  For others, however, they may intentionally (or not) leave out some test case that you should be aware of. If you are not completely sure that your code is correct, test extensively, and try to come up with your own test cases as well. You may even be able to use them in the challenge phase.

Solving a problem
Now we'll walk through a simple problem and dissect it, bit by bit.

Have a look at BettingMoney, the SRM 191 Division 2 Easy. First we identify the parts of this problem. In the statement itself, we first have the situation behind the problem – gambling. Then we have a little bit of background information about the betting itself. Then, we have a description of the input – data types, variable names, and what they represent. After this we have the task: to determine what the net gain is for the day and return the amount in cents.

Also note the two explanatory paragraphs at the end; the first provides an example of the input format and types, and the second gives a completely worked example, which should be extremely helpful to your understanding.

The definition section is uninteresting, but it is there for completeness' sake.

The notes for this problem are fairly comprehensive. In terms of background information, you might not know that there are 100 cents in a dollar. And in terms of clarification, there is explicit confirmation that the return value may in fact be negative, and that the margin of victory (the variable finalResult) is all that matters when deciding which payoff to make.

The constraints are fairly straightforward. The input arrays will contain the same number of elements, between 1 and 50, inclusive. (50 is a long-standing topcoder tradition for input sizes). finalResult will be between 0 and that same size minus one (which means, if you give it a little thought, that someone will win their bet). Each element of each array will be between 0 and 5000, inclusive. This is most likely to make sure that integer arithmetic will do the job just fine.

Finally, there's the examples section. Often, the problem statement section will contain an annotated example case, which will become example case 0. Then there are a couple of other example cases, some with explanation and some without. Also note that one of the examples tests for negative return values, to supplement the notes.

A More Complicated Example
Now have a look at Poetry, the SRM 170 Div 2 Hard. In this case, you may not be able to actually solve this in the time allotted. That's ok – the emphasis should first be on understanding what the problem says, even if you can't code it in time.

The first section tells you immediately what you want to do – you'll be given a poem, and you will have to determine what its rhyme scheme is. The rest of the section clarifies what this actually means, in bottom-up fashion (from simpler concepts to more complicated ones). It defines what a legal word is and how to extract words from a poem, and then it defines what it means when two words rhyme – that their ending patterns are equal. The concept of ending pattern is then defined. After all this, we find out what it means to have two lines of the poem rhyme: their last words have to rhyme. Finally, (whew!) we are told how to actually construct the rhyme scheme and in what format to return it.

This is a problem where a lot of terms need to be defined to get to the heart of things, and so all the definitions deserve at least a couple of read-throughs, especially if you're not sure how they all fit together.

The next section is the problem definition section, just for reference. Then there is a single note that clarifies a point that may have been overlooked when it was stated in the problem statement itself: that blank lines will be labeled with a corresponding space in the rhyme scheme.

The constraints are fairly standard for topcoder problems: there will be between 1 and

50 lines in the poem, and each line will contain between 0 and 50 characters. The only allowable characters in the poem will be spaces and letters, and there will be only legal words in poem.

Finally, there are a number of examples. Usually, problems which are trickier or which have more complex problem statements will have more examples, to clarify at least some of the finer points of the problem statement. Again, this doesn't mean that passing the example cases given is equivalent to having a completely correct solution, but there is a higher chance that you can catch any bugs or trivial mistakes if there are more examples that you know the answers to.

Try it Yourself
Listed below are a number of additional problems, grouped roughly by difficulty of comprehension. Try them for yourself in the topcoder Arena Practice Rooms. Even if you can't solve them, at least work on figuring out what the problem wants by breaking it down in this manner.

Mentioned in this writeup:
SRM 203 Div 2 Easy – UserName
SRM 191 Div 2 Easy – BettingMoney
SRM 203 Div 1 Easy – MatchMaking
SRM 170 Div 2 Hard – Poetry

Similar tasks:
SRM 146 Div 2 Easy – Yahtzee
SRM 200 Div 2 Easy – NoOrderOfOperations
SRM 185 Div 2 Easy – PassingGrade
SRM 155 Div 2 Easy – Quipu
SRM 147 Div 2 Easy – CCipher
SRM 208 Div 1 Easy – TallPeople
SRM 173 Div 1 Easy – WordForm
SRM 162 Div 1 Easy – PaperFold

More challenging tasks:
SRM 197 Div 2 Hard – QuickSums
SRM 158 Div 1 Hard – Jumper
SRM 170 Div 1 Easy – RecurrenceRelation
SRM 177 Div 1 Easy – TickTick
SRM 169 Div 2 Hard – Twain
SRM 155 Div 1 Med – QuipuReader