TopCoder Training Camp >> Tutorials >> TopCoder Survival Kit

Author: Stefan Pochmann
Last updated: March 23, 2002

Note: I've written this a long time ago, so maybe I have changed my mind about some things in the meantime. Moreover, some of this stuff has already made its way into my other tutorials, for example the API links. Nevertheless, I believe that it still contains some valuable advice, especially for newbies.

This page contains some more or less general rules that I came up for myself to be better in TopCoder programming contests. However, most of them should apply to other programming contests or even real life, as well.

But first a general remark about why there's no good reason to not participate in the contests: Even if you lose, because you make stupid mistakes like the ones I recently made, you still win! Do you think I would've written this survival kit without having mistakes as a reason? Do you think I would've thought so much about this stuff? Making mistakes was in the long term even better, because now I've learned more. You simply can't lose!

Note: If I'm sometimes rude here, please don't take it personally. I'm screaming at myself. This page is more like a written soliloquy for me.

General problem solving structure

Here are some donts:

- Don't submit your program before having tested it. Exception: You just finished coding and it's 5 seconds until the end of the contest. This is the only exception. Remember: In TopCoder you don't have a second chance to submit.
- Don't program your algorithm before you have designed and tested it. I tend to fly over the problem statement, not reading everything, because I know what the want anyway. Right? Wrong! Usually I miss an important constraint and go for the wrong goal.
- Don't design your algorithm before you have understood the problem. Several times I had not fully understood the problem. I just thought "Oh well, this is similar to the XYZ problem that I already know". And then I solved the wrong problem. Great idea, dumbass!
- You have not understood the problem before you have read the whole description, and analyzed the examples. If you don't read everything, you'll miss the most important sentence. I promise!

Or in chronological order, this is what you should do:

1. Read the problem description. Write down all important things: What is the data? What is the unknown? What are the constraints?
2. Analyze the examples. More on examples later in the survival kit.

3. Make sure you have understood the problem and know all the constraints. What is the goal? Do the examples all make sense?

4. Design your algorithm and evaluate it. Does it work for the examples? Is it fast enough? Note: If possible, this phase is on paper!

5. Program your algorithm. Now that's what you wanted to do all the time, right? Now you're allowed to.

6. Test your program with the given examples. Come up with new examples if the given ones are not enough. Remember: In TopCoder you don't have a second chance to submit!

Last rule: there's always an exception to (almost) every rule. For example, a simple test program can sometimes help you to understand the problem. In that case, you can change the fixed process above.

Examples

Read all examples. At the very least glance over them. But do glance over them. Note that this is not the same as skipping them. I divide examples as follows: trivial vs non-trivial, full vs magic.

Trivial examples are ... well ... trivial. They are so small and simple, that you can't learn much from them. But remember Murphy's Law: as soon as you skip the most simple example, it will later turn out to be the most important one. For example, the one that highlights a special case where n=0.

Magic examples are so big that you can't understand the answer, which of course only includes the return value. In contrast, a full example is understandable, either because it's small enough or because the steps that have lead to the solution are given.

For example, recently a problem involved finding a somewhat optimal submultiset of a multiset. I of course skipped all good examples and went directly to the biggest one. After all, the big ones are the only interesting ones, right? Wrong! Totally wrong! For that example, only the pure answer was given, in that case the sum of all elements in the submultiset. But in the previous example, which was way smaller (but still quite large), not only the return value, but also the submultiset that achived that result was given. Now guess which example can help understanding the problem better.

Usually TopCoder provides some trivial examples, then some that point out each important goal and then some large magic ones. These people are really nice, so use what they give you! Examples sometimes provide not only the best way to understand a problem, but the only way. Do not underestimate the value of a good example!

Sentinels

Sentinels are the second best invention of humankind, only toilet paper is more useful. Ok, I overdo it a bit. What is a sentinel? It's an element that you place at a good strategic position so that you can save some tests. Here's an example with a function that determines whether a key element appears somewhere in an array. The version without sentinel:

```
bool find ( int a[], int n, int key ) {
    for( int i=0; i<n; ++i )
        if( a[i] == key )
            return true;
    return false;
}
```

The same function with a sentinel:

```
bool find ( int a[], int n, int key ) {
    a[n] = key;
    for( int i=0; ; ++i )
        if( a[i] == key )
            return (i<n);
}
```

Here a sentinel is introduced by putting the key we search for after the end of the array we search in. This ensures that we *will* eventually find the element. And then when we find it, we only have to check whether we found a real element or just the sentinel. (Thanks **isv** for making me realize I should explain this example).

Of course this only works if we can use the array spot `a[n]` to place our sentinel there. But usually you can just make your array larger than you need and the remaining entries are not used, so that's no problem. What's the point of using a sentinel? There are two:

- A sentinel can make the program faster. In the above example, we save the test "`i<n`" in every loop iteration. However, this is nice in real life, but will not be the reason why you get your program accepted in TopCoder. So forget about this reason!

- A sentinel can make the program simpler. This is not really true for the above example, but you'll see an example in the discussion on functional sentinels. Making a program simpler is good! It's exactly what you want for a programming contest. Why? Because it saves time and you don't get as many chances to add bugs.

Look into "Introduction to Algorithms" (CLRS) for a nice example of sentinels for double-linked lists, where the program really got simpler. If you don't have the book, buy it. It's the new bible for algorithm addicts like us.

Sentinels can appear in many forms. One more example (also found in CLRS): In mergesort, you merge two parts. The common merging always has to look if one part is already finished. By placing a new largest value behind both parts, you can omit this test.

Functional Sentinels

In a recent contest, one problem was to determine the winner of a Connect Four game (the real question was somewhat more inflated with details, but that's essentially the task).

What did clever Stefan do? I made the array for the field larger than necessary. This way, I got sentinels to the right and to the bottom of the valid area. To test whether there were four consecutive 1's somewhere, I just started from every position in the real field and walked in each of the three directions (down, right, down+right). I never had to test whether I'm still in the valid range, since elements outside the valid range were no 1's and so my loop stopped at them anyway.

Here's the problem: I forgot about the fourth direction, up+right. Even if I had noticed this mistake before submitting it, it would've involved some thinking to fix it. What would be a fix? Think about it for a moment before you read on.....Ok, one possibility would be to start the real field in the second row of the array, not the first one. This would give me extra sentinels above the field. But then I start indexing at 1 again, which caused a bug in another contest.

Here's what I would do now: Make the array exactly as large as needed. Then, never ever directly access its elements. Instead, use the this function:

```
int f ( int i, int j ) {
    return (i>=0 && i<n && j>=0 && j<m) ? field[i][j] : -1;
}
```

This way I automatically get sentinels in every invalid position. I'll never have to worry.

Wait a minute! This not only undoes the speed gain we got from using sentinels, but adds an extra function call overhead, too! How stupid would it be to use this? It's not stupid, dammit! Just as the speed gain of sentinels won't make a difference between fast enough and too slow, this speed loss won't make one. After all, few people will use sentinels, so our overhead is only for the function call. But this will not push it over the limit! The problem setters are nice guys. Usually you don't have a chance to get even close to the time limit, unless you're in the totally wrong complexity class.

If you're really worried, then in C++ make "`f`" a macro instead of a function (or inline? I never used that, don't know). This will also remove the need for having n, m and field globally accessible.

But consider the gain! This will save you a lot of time and headaches! You can still start with index 0, and you don't have to make sure that subsequent calls need an initialization of the array (imagine going from a large field to a smaller one. Then to the right and under the current field, there might be the old values instead of sentinels).

Some short advice

- You can always assume that the input will have the specified the constraints. If they say that the array contains at least one element, then this will be ensured. If somebody wants to challenge you with invalid input, he won't be able to. TopCoder checks the input before testing your program on it.

- Use array indices consistently. Recently I once again used the indices 0..5 in one part of my program and 1..6 in another part. Generally, this is a bad idea, don't you think? Yes it is, you idiot! Why do I have to tell you this over and over again (Remember that I'm talking to myself).

  There are only two reasons to start from 1: The problem statement starts from 1. Bullshit reason! What are you, a whiny coward? Just shift everything by one. The other reason: It actually makes sense, for example if you want to place a sentinel at position 0. But in this case, for sure you won't accidentially forget that you're starting from 1, because the sentinel plays a key part in your code.

- If the problem defines a new type, then make it a new type. Write a class for it. For example, if they come up with "Trilean" values (true, false or maybe), then create a `class Trilean` with all needed operations. Using (say) an `int` to represent such a value will only spread the code for the operations all over the program. Represent it as an int inside the Trilean object.

- Use constants with cool names. In the Trilean example, make up constants TRUE, FALSE and MAYBE. I've actually seen this in lots of programs of higer-rated coders. It can also make the program shorter, but the key benefit is that you can't forget that 2 represents TRUE, not FALSE.

- If the problem describes a black box data structure and its behaviour and asks you to implement that, then create an actual class for it. This will make things so much clearer. Well, of course it always depends on how much overhead you get, same for the Trilean class. But often it's worth it.

- Look at the other coders. How far are they? In the last contest, I was so freaking fast, you could'nt believe it. I had solved all three problems before anyone else in my room had solved his/her second one. In fact, I was way faster than the whole division! Is that great or what? It's "or what"! It's stupid! Instead, I should've taken some time to test my last program better. It was the Connect Four problem, so you already know that I screwed it.

- In TopCoder, inputs tend to be small. So why go for the heavy 30 line solution that gives you O(n lg n) when n never exceeds 50 and there's a 5 line solution with O(n^3)? Don't underestimate the power of modern computers and small inputs.

- Have your favourite useful websites ready. Some suggestions: Java 1.4 API - C++ Reference - C++ Reference C++ Reference

- For hard algorithmic problems, have your favourite useful books ready. Some suggestions:

  - Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)
  - The Algorithm Design Manual (Steven Skiena)
  - Algorithms in C++ (Parts 1-4) (Robert Sedgewick)
  - Algorithms in C++ (Part 5) (Robert Sedgewick) Note that most reviews are for an old version.

  Unfortunately, the 75 minutes of a TopCoder contest are pretty short. So only look into a book if you know what's in it (you don't have the time to read a book) or if you're really desperate.

Stefan Pochmann
Last modified: Thu Dec 12 14:23:35 MET 2002