- Competitions
- TopCoder Networks
- Events
- Statistics
- Tutorials
  - Overview
  - Algorithm Tutorials
  - Software Tutorials
  - Marathon Tutorials
  - Wiki
- Forums
- Surveys
- My TopCoder
- Help Center
- About TopCoder

UML TOOL

Member Search:
Handle: Go
Advanced Search

Algorithm Tutorials

Planning an Approach to a TopCoder Problem

Archive
Printable view
Discuss this article
Write for TopCoder

By leadhyena_inran
TopCoder Member

Planning an approach is a finicky art; it can stump the most seasoned coders as much as it stumps the newer ones, and it can be extremely hard to put into words. It can involve many calculations and backtracks, as well as foresight, intuition, creativity, and even dumb luck, and when these factors don't work in concert it can inject a feeling of helplessness in any coder. Sometimes it's this feeling of helplessness that discourages coders from even attempting the Div I Hard. There are even coders that stop competing because they abhor that mental enfeeblement that comes with some problems. However, if one stays diligent, the solution is never really out of the mind's reach. This tutorial will attempt to flesh out the concepts that will enable you to pick an approach to attack the problems with a solid plan.

Pattern Mining and the Wrong Mindset

It is easy to fall into the trap of looking at the algorithm competition as a collection of diverse yet classifiable story problems. For those that have done a lot of story problems, you know that there are a limited number of forms of problems (especially in classes where the professor tends to be repetitious), and when you read a problem in a certain form, your mind says, "Oh, this is an X problem, so I find the numbers that fit the problem and plug and chug." There are many times when this kind of pattern mining pays off; after a number of TopCoder Single Round Matches, most coders will recognize a set of common themes and practice against them, and this method of problem attack can be successful for many matches.

However, this approach is perilous. There are times when you skim the problem statement and assume it's of type Q, then start coding and discover that your code passes none of the examples. That's when you reread the problem and find out that this problem is unique to your experience. At that point, you are paralyzed by your practice; being unable to fit any of your problem types to the problem you are unable to proceed. You'll see this often when there's a really original problem that comes down the pipe, and a lot of seasoned coders fail the problem because they are blinded by their experience.

Pattern mining encourages this kind of mindset that all of the problem concepts have been exhausted, when in reality this is impossible. Only by unlearning what you have learned (to quote a certain wise old green midget) and by relearning the techniques of critical thought needed to plan an approach can your rating sustainably rise.

Coding Kata

Here's your first exercise: take any problem in the Practice Rooms that you haven't done. Fight through it, no matter how long it takes, and figure it out (use the editorial from the competition as a last resort). Get it to pass system tests, and then note how long you took to solve it. Next, clear your solution out, and try to type it in again (obviously cutting and pasting will ruin the effect). Again, get it to pass system tests. Note how long it took you to finish the second time. Then, clear it out and do the problem a third time, and again get it to pass system tests. Record this final time.

The time it takes for your first pass is how long it takes you when you have no expectations of the problem and no approach readily in mind. Your time on the second pass is usually the first time minus the amount of time it took you to understand the problem statement. (Don't be surprised at the number of bugs you'll repeat in the second pass.) That final recorded time is your potential, for you can solve it this fast in competition if you see the correct approach immediately after reading it. Let that number encourage you; it really is possible to solve some of these problems this quickly, even without super fast typing ability. But what you should also learn from the third pass is the feeling that you knew a working strategy, how the code would look, where you would tend to make the mistakes, and so on. That's what it feels like to have the right approach, and that feeling is your goal for future problems in competition.

In most martial arts, there's a practice called kata where the martial artist performs a scripted series of maneuvers in order, usually pretending to defend (or sometimes actually defending) against an onslaught of fighters, also scripted to come at the artist predictably. At first this type of practice didn't make any sense, because it didn't seem realistic to the chaotic nature of battle. Furthermore it seems to encourage the type of pattern mining mentioned in the previous section. Only after triple-coding many problems for a while can one comprehend the true benefit of this coding kata. The kata demonstrates to its practitioners the mental experience of having a plan, encouraging the type of discipline it takes to sit and think the problem through. This plan of attack is your approach, and it carries you through your coding, debugging, and submission.

Approach Tactics

Now that you know what an approach feels like and what its contents are, you'll realize that you know a lot of different types of these approaches. Do you give them names? "Oh, I used DP (dynamic programming) on that problem." "Really, I could have done that one greedy?" "Don't tell me that the brute-force solution would have passed in time." Really, the name you give an approach to a problem is a misnomer, because you can't classify every problem as a type like just greedy or just brute-force. There are an infinite number of problem types, even more solution types, and even within each solution type there are an infinite number of different variations. This name is only a very high level summary of the actual steps it takes to get to the solution.

In some of the better match editorials there is a detailed description of one approach to solving the code. The next time you look at a match summary, and there is a good write-up of a problem, look for the actual steps and formation of the approach. You start to notice that there is a granularity in the steps, which suggests a method of cogitation. These grains of insight are approach tactics, or ways to formulate your approach, transform it, redirect it, and solidify it into code that get you closer to the solution or at least point you away from the wrong solution. When planning your approach, the idea is that you will use whatever approach tactics are at your disposal to

decide on your approach, the idea being that you are almost prewriting the code in your head before you proceed. It's almost as if you are convincing yourself that the code you are about to write will work.

Coders with a math background may recognize this method of thinking, because many of these approach tactics are similar to proof writing techniques. Chess players may identify it with the use of tactics to look many moves ahead of the current one. Application designers may already be acquainted with this method when working with design patterns. In many other problem solving domains there is a similar parallel to this kind of taxonomy.

To practice this type of critical thinking and to decide your preferences among approach tactics, it is very useful to record the solutions to your problems, and to write up a post-SRM analysis of your own performance. Detail in words how each of your solutions work so that others could understand and reproduce the approach if they wanted to just from your explanations. Not only will writing up your approaches help you to understand your own thoughts while coding, but this kind of practice also allows you to critique your own pitfalls and work on them in a constructive manner. Remember, it is difficult to improve that which you don't understand.

Breaking Down a Problem

Let's talk about one of the most common approach tactics: breaking down a problem. This is sometimes called top-down programming: the idea is that your code must execute a series of steps in order, and from simple decisions decide if other steps are necessary, so start by planning out what your main function needs before you think about how you'll do the subfunctions. This allows you to prototype the right functions on the fly (because you only code for what you need and no further), and also it takes your problem and fragments it into smaller, more doable parts.

A good example of where this approach is useful is in MatArith from Round 2 of the 2002 TopCoder Invitational. The problem requires you to evaluate an expression involving matrices. You know that in order to get to the numbers you'll need to parse them (because they're in String arrays) and pass those values into an evaluator, change it back into a String array and then you're done. So you'll need a print function, a parse function and a new calc function. Without thinking too hard, if you imaging having all three of these functions written already the problem could be solved in one line:

```
public String[] calculate(String[] A, String[] B, String[] C, String eval){
   return print(calc(parse(A),parse(B),parse(C),eval));
}
```

The beauty of this simplest approach tactic is the guidance of your thoughts into a functional hierarchy. You have now fragmented your work into three steps: making a parse function, a print function, and then a calc function, breaking a tough piece of code into smaller pieces. If you break down the code fine enough, you won't have to think hard about the simplest steps, because they'll become atomic (more on this below). In fact the rest of this particular problem will fall apart quickly by successive partitioning into functions that multiply and add the matrices, and one more that reads the eval statement correctly and applies the appropriate functions.

This tactic really works well against recursive problems. The entire idea behind recursive code is that you are breaking the problem into smaller pieces that look exactly like the original, and since you're writing the original, you're almost done. This approach tactic also plays into the hands of a method of thinking about programs called functional programming. There are several articles on the net and even a TopCoder article written by radeye that talk more about this concept in depth, but the concept is that if properly fragmented, the code will pass all variable information between functions, and no data needs to be stored between steps, which prevents the possibility of side-effects (unintended changes to state variables between steps in code) that are harder to debug.

Plan to Debug

Whenever you use an approach you should always have a plan to debug the code that your approach will create. This is the dark underbelly of every approach tactic. There is always a way that a solution may fail, and by thinking ahead to the many ways it can break, you can prevent the bugs in the code before you type them. Furthermore, if you don't pass examples, you know where to start looking for problems. Finally, by looking for the stress points in the code's foundation, it becomes easier to prove to yourself that the approach is a good one.

In the case of a top-down approach, breaking a problem down allows you to isolate sections of the code where there may be problems, and it will allow you to group tests that break your code into sections based on the subfunction they seem to exploit the most. There is also an advantage to breaking your code into functions when you fix a bug, because that bug is fixed in every spot where the code is used. The alternative to this is when a coder copy/pastes sections of code into every place it is needed, making it harder to propagate a fix and makes the fix more error prone. Also, when you look for bugs in a top-down approach, you should look for bugs inside the functions before you look between the calls to each function. These parts make up a debugging strategy: where to look first, how to test what you think is wrong, how to validate pieces and move on. Only after sufficient practice will a debugging strategy become more intuitive to your method of attack.

Atomic Code

If you arrive at a section of code that you cannot break down further this is atomic code. Hopefully you know how to code each of these sections, and these form the most common forms of atomic code. But, don't be discouraged when you hit a kernel of the problem that you don't know how to code; these hard-to-solve kernels are in fact what make the problem interesting, and sometimes being able to see these in advance can make the big difference between solving the problem early with the right approach and heading down the wrong path with the wrong approach, wasting a lot of time in the process.

The most common type of atomic code you'll write is in the form of primitives. I've always been a proponent of knowing the library of your language of choice. This is where that knowledge is of utmost importance. What better way to save yourself time is there in both planning your approach and coding your solution when you know that a possibly difficult section of your code is in fact atomic and solved using a library function or class?

The second type of atomic code you'll write are what I call language techniques. These are usually snippets of code committed to memory that perform a certain operation in the language, like locating the index of the first element in an array with the minimum value, or parsing a String into tokens separated by whitespace. These techniques are equally essential to planning an approach, because if you know how to do these fundamental operations intuitively, it makes more tasks in your search for a top-down approach atomic, thus making the

search for the right approach shorter. In addition, it makes the segments of the code in these atomic segments less error prone. Furthermore, if you are asked to perform a task similar to one that you already know a language technique for, it makes it much easier to mutate the code to fit the situation (for example: searching for the index of the first maximal element in an array based on some heuristic is easy if you already know how to type up similar tasks). Looking for these common language techniques should become an element of your daily practice, and any atomic code should fly off your fingers as soon as you think about it.

As an aside, I must address the use of code libraries. I know that this is a contested topic, and many successful coders out there make use of a (sometimes encyclopedic) library as a pre-inserted segment of code before they start coding. This is totally legal (although changes to the rules after the 2004 TopCoder Open may affect their future legality), and there are obvious advantages to using a library, mainly through the ability to declare more parts of your top-down approach atomic, and by being able to more quickly construct bottom-up fragments of code (as discussed below). It is my opinion, however, that the disadvantages of using library code outweigh the advantages. On a small note, library code executed through functions can sometimes slow your coding, because you have to make the input match the prototype of the code you're trying to use. Library code is mostly non-mutatable, so if your library is asked to do something that isn't expressly defined, you find yourself fumbling over a language technique or algorithm that should already be internalized. It is also possible that your library code isn't bug-free, and debugging your library mid-competition is dangerous because you may have to propagate that change to code you've already submitted and also to the template before you open any more problems. Also, library use is not allowed in onsite competition. Finally, the use of library code (or macros for that manner) get you used to leaning on your library instead of your instincts of the language, making the use of normal primitives less intuitive and the understanding of other coder's solutions during challenge phase not as thorough. If used in moderation your library can be powerful, but it is not the ultimate weapon for all terrain.

There may be a point where you hit a piece of atomic code that you are unable to fragment. This is when you have to pull out the thinking cap and start analyzing your current approach. Should I have broken up the tasks differently? Should I store my intermediate values differently? Or maybe this is the key to the problem that makes the problem hard? All of these things must be considered before you pound the keys. Even at these points where you realize that you're stuck, there are ways to manipulate the problem at hand to come to an insight on how to proceed quickly, and these ways comprise the remaining approach tactics.

...continue to Section 2