Planning an Approach to a Topcoder Problem: Part 2

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

...read Section 1

Bottom Up Programming
This technique is the antithesis to breaking down a program, and should be the first thing you start doing when you get stuck. Bottom-up programming is the process of building up primitive functions into more functional code until the solution becomes as trivial as one of the primitives. Sometimes you know that you'll need certain functions to form a solution and if these functions are atomic or easy to break down, you can start with these functions and build your solution upward instead of breaking it down.

In the case of MatArith, the procedure to add and multiply matrices was given in the problem statement making it easy to follow directions and get two functions to start with. From there you could make a smaller evalMult function that multiplied matrices together using a string evaluation and variable names, then a similar evalAdd that treats each term as a block and you have an approach to solve the problem.

In general, it's a very good strategy to code up any detailed procedure in the problem statement before tackling the actual problem. Examples of these are randomizer functions, any data structures you're asked to simulate, and any operations on mathematical objects like matrices and complex numbers. You'll find that by solving these smaller issues and then rereading the problem statement that you will understand what needs to be done much better. And sometimes, if you're really stuck, it doesn't hurt to write a couple atomic pieces of code that you know that you'll need in order to convince your mind to break down the problem towards those functions. As you can see, your path to the right approach need not be linear as long as it follows your train of thought.

Also, in case of a hidden bug, keep in mind that any code that you write using this approach tactic should be scanned for bugs before your top-down code, because you tend to write this code first and thus at a stage where you understand the problem less than when you finish the code. This is a good rule of thumb to follow when looking for errors in your code; they usually sit in the older sections of the code, even if older is only decided by minutes or even seconds.

Brute Force
Any time the solution requires looking for an optimal configuration or a maximal number or any other choice of one of a finite set of objects, the simplest way to solve

the problem is to try all configurations. Any time the solution requires calculating a massive calculation requiring many steps, the best way to solve it is to do every calculation as asked for in the problem. Any time the problem asks you to count the number of ways something can be done, the best way to solve it is to try every way and make a tally. In other words, the first approach to consider in any possibly time-intensive problem is the most obvious one, even if it is horribly inefficient.

This approach tactic, called brute force, is so called because there is no discerning thought about the method of calculation of the return value. Any time you run into this kind of an optimization problem the first thing you should do is try to figure in your head the worst possible test cases and if 8 seconds is enough time to solve each one. If so, brute force can be a very speedy and usually less error prone approach. In order to utilize brute force, you have to know enough about the programming environment to calculate an estimate how much time any calculation will take. But an estimate is just a guess, and any guess could be wrong. This is where your wisdom is forced to kick in and make a judgment call. And this particular judgment call has bitten many a coder that didn't think it could be done brute force and couldn't debug a fancier approach, and likewise those that didn't figure correctly the worst of the cases to be tested for time.

In general, if you can't think of a way to solve the problem otherwise, plan to use brute force. If it ends up that you are wrong, and there is a test case that takes too long, keep the brute force solution around, and while recoding the more elegant solution, use the brute-force solution to verify that your elegant code is correct in the smaller cases, knowing that its more direct approach is a good verification of these cases (being much less error-prone code).

A Place for Algorithms

Well-known and efficient algorithms exist for many standard problems, much like basic approaches exist for many standard word problems in math, just like standard responses exist for most common opening moves in chess. While in general it's a bad idea to lean heavily upon your knowledge of the standard algorithms (it leads down the path of pattern mining and leaves you vulnerable to more original problems), it's a very good idea to know the ones that come up often, especially if you can apply them to either an atomic section of code or to allow them to break your problem down.

This is not the place to discuss algorithms (there are big books to read and other tutorials in this series to follow that will show you the important stuff), but rather to discuss how algorithms should be used in determining an approach. It is not sufficient to know how to use an algorithm in the default sense; always strive to know any algorithms you have memorized inside and out. For example, you may run into a problem like CityLink (SRM 170 Div I Med), which uses a careful mutation of a basic

graph algorithm to solve in time, whereby just coding the regular algorithm would not suffice. True understanding of how the algorithm works allows the insight needed to be able to even conceive of the right mutation.

So, when you study algorithms, you need to understand how the code works, how long it will take to run, what parts of the code can be changed and what effect any changes will have on the algorithm. It's also extremely important that you know how to code the algorithm by memory before you try to use it in an approach, because without the experience of implementing an algorithm, it becomes very hard to tell whether your bugs are being caused by a faulty implementation or faulty input into the implementation. It's also good to practice different ways to use the algorithms creatively to solve different problems, to see what works and what doesn't. Better for an experiment with code to fall flat on its face in practice than during a competition. This is why broad-based algorithmic techniques (like divide-and-conquer, dynamic programming, greedy algorithms) are better to study first before you study your more focused algorithms because the concepts are easier to manipulate and easier to implement once you understand the procedure involved.

Manipulating the Domain

This situation will become more and more familiar: you find yourself trudging through the planning stages of a problem because of the sheer amount of work involved in simulating the domain of the problem. This may be due to the inappropriateness of the presented domain, and there are times when manipulating the domain of the problem to something more convenient will create an easier or more recognizable problem. The classic example of this is the game of Fifteen (used as a problem in SRM 172). In the game of Fifteen, you have numbers from 1 to 9 of which you may claim one per turn, and if exactly three of your numbers add to 15 before exactly three of your opponent's numbers adds to 15, then you win. For this problem, you can manipulate the domain by placing the numbers in the configuration of a 3×3 magic square (where every row, column, and diagonal add up to the same sum, in this case 15). Instantly you realize that the game of Fifteen is just the game Tic-Tac-Toe in disguise, making the game easier to play and program a solution for, because the manipulation of the domain transformed the situation of a game where you have no prior knowledge into one where you have a lot more knowledge. Some mathematicians think of this as the ABA^{-1} approach, because the algebra suggests the proper process: first you transform the domain, then you perform your action, then (the A^{-1}) you reverse your transformation. This approach is very common in solving complex problems like diagonalizing matrices and solving the Rubik's Cube.

Most commonly this approach tactic is used to simplify basic calculations. A good example of this type of approach is HexagonIntersections from SRM 206. In this

problem it was needed to find the number of tiled hexagons that touched a given line. The problem became much easier if you "slanted" the grid by transforming the numbers involved so that the hexagons involved had sides parallel to the x and y axis and the problem still had the same answer, thereby simplifying calculations.

Extreme care must be taken while debugging if you manipulate the domain. Remember that the correct procedure to domain manipulation is to first manipulate the domain, then solve the problem, and then correct the domain. When you test the code, remember that either the domain must be properly reversed by the transformation before the result is returned, or the reversal must not affect the answer. Also, when looking at values inside the domain manipulation, remember that these are transformed values and not the real ones. It’s good to leave comment lines around your transformed section of code just to remind yourself of this fact.

Unwinding the Definitions

This approach tactic is an old mathematician’s trick, relating to the incessant stacking of definitions upon definitions, and can be used to unravel a rather gnarly problem statement to get at the inner intended approach to the problem. The best way to do this is with code. When you read the definition of something you have never encountered before, try to think how you would code it. If the code asks you to find the simplest *grozmojt* in a set of integers, first figure out how your code would verify that something was a *grozmojt* and then figure out how to search for it, regardless if you even need to verify that something was a *grozmojt* in the solution. This is very similar to the bottom-up programming above, but taken at the definition level instead of the procedural one.

Simulation problems fall under similar tactics, and create one of those times when those predisposed to object oriented coding styles run up the scores. The best way to manage a simulation problem is to create a simulation object that can have actions performed on it from a main function. That way you don’t worry if you passed enough state into a given function or not; since all of the information in the simulation is coming along with you, the approach becomes very convenient and reaches atomic code very quickly. This is also the correct approach to take if an algorithm needs to be simulated to count the steps needed in the algorithm (like MergeSort) or the number of objects deallocated in the execution of another algorithm (like ImmutableTrees). In these situations, the elegance of the code is usually sacrificed in the name of correctness and thoroughness, also making the approach easier to plan ahead for.

The Problem is Doable

An old geometry puzzle goes like this: you have a pair of concentric circles and the only length you are given is the length of a chord of the outer circle (call the chord length x) that is tangent to the inner circle, and you are asked for the area between the

circles. You respond: "Well, if the problem is doable then the inner circle's radius is irrelevant to the calculation, so I'll declare it to be 0. Because the area of the inner circle is 0, or degenerates to the center of the outer circle, the chord of the outer circle passes through the center and is thus the diameter, and thus the area of the outer circle is $Pi(x/2)^2$." Note that a proper geometric proof of this fact is harder to do; the sheer fact that a solution exists actually makes the problem easier. Since the writer had to write a solution for the problem, you know it's always solvable, and this fact can be used to your advantage in an SRM.

This approach tactic broadens into the concept that the writer is looking for a particular type of solution, and sometimes through edits of the original problem statement this approach is given away (especially if the original problem is considered too hard for the level it's at). Look for lowered constraints like arrays of size 20 (which many a seasoned coder will tell you is almost a codeword that the writer is looking for a brute force solution), or integers limited to between 1 and 10000 (allowing safe multiplication in ints without overflow). By leaning on the constraints you are acting similarly to the situation above, by not allowing the complexities of the problem that were trimmed off by the constraints to complicate your approach.

Sometimes the level of the problem alone will give a hint to what solution is intended. For example, look at FanFailure (from SRM 195 Div I Easy). The problem used the language of subsets and maximal and minimal, so you start to think maybe attack with brute force, and then you see the constraints opened up to 50 for the array size. 2^{50} distinct subsets rules out brute force (better to find this out in the approach than in the code, right?) and you could look to fancier algorithms... but then you realize that this is a Div I Easy and probably isn't as hard as it looks so you think through the greedy algorithm and decide that it probably works. This choice wouldn't have been so obvious had it not been a Div I Easy.

Keep in mind that these invisible cues are not objective and can't be used to reason why an approach will work or not; they are there only to suggest what the writer's mind was thinking. Furthermore, if the writer is evil or particularly tricky, cues of this nature may be red herrings to throw these tactics astray. As long as you temper this approach tactic with solid analysis before you go on a wild goose chase, this "circular reasoning" can be used to great advantage.

Case Reduction

Sometimes the simplest problems to state are the ones that provide the most difficulty. With these types of problems it's not unusual that the solution requires that you break up the problem not into steps but into cases. By breaking a problem up into cases of different sets of inputs you can create subproblems that can be much easier to solve. Consider the problem TeamPhoto (SRM 167 Div I Medium). This problem is simple to

state, but abhorrent to solve. If you break up the problem into a series of cases, you find that where the entire problem couldn't alone be solved by a greedy algorithm, each of the different cases could be, and you could take the best case from those optimal configurations to solve the problem.

The most common use of case reduction involves removing the boundary cases so that they don't mess up a naï¿½ve solution. A good example of this is BirthdayOdds (SRM 174 Div I Easy); many people hard coded if(daysInYear==1)return 2; to avoid the possible problems with the boundary case, even if their solution would have handled it correctly without that statement. By adding that level of security, it became easier to verify that the approach they chose was correct.

Plans Within Plans

As illustrated above, an approach isn't easily stated, and is usually glossed over if reduced to a one-word label. Furthermore, there are many times when there exist levels to a problem, each of which needs to be solved before the full solution comes to light. One clear example of this is the problem MagicianTour (SRM 191 Div I Hard). There are definitely two delineated steps to this problem: the first step requires a graph search to find all connected components and their 2-coloring, and the second step requires the DP knapsack algorithm. In cases like this, it's very helpful to remember that sometimes more than one approach tactic needs to be applied to the situation to get at the solution. Another great example is TopographicalImage (SRM 209 Div I Hard) which asks for the lowest angle that places a calculated value based on shortest path under a certain limit. To solve, note that looking for this lowest value can be approached by binary search, but there are plans within plans, and the inner plan is to apply Floyd-Warshall's All Pairs Shortest Paths algorithm to decide if the angle is satisfactory.

Remember also that an approach isn't just "Oh, I know how to break this down... Let's go!" The idea of planning your approach is to strategically think about: the steps in your code, how an algorithm is to be applied, how the values are to be stored and passed, how the solution will react to the worst case, where the most probable nesting places are for bugs. The idea is that if the solution is carefully planned, there is a lower chance of losing it to a challenge or during system tests. For each approach there are steps that contain plans for their steps.

Tactical Permutation

There is never a right approach for all coders, and there are usually at least two ways to do a problem. Let's look at an unsavory Division One Easy called OhamaLow (SRM 206 Div I Easy). One of the more popular ways to approach this problem is to try all combinations of hands, see if the hand combination is legal, sort the hand combination, and compare it to the best hand so far. This is a common brute-force

search strategy. But it's not the entire approach. Remember that there are plans within plans. You have to choose an approach on how to form each hand (this could be done recursively or using multiple for-loops), how to store the hands (int arrays, Strings, or even a new class), and how to compare them. There are many different ways to do each of these steps, and most of the ways to proceed with the approach will work. As seen above as well as here, there are many times where more than one approach will do the job, and these approaches are considered permutable. In fact, one way to permute the top-level brute-force search strategy is to instead of considering all possible constructible hands and picking the best one, you can construct all possible final hands in best to worst order and stop when you have found one that's constructible. In other words you take all 5 character substrings of "87654321" in order and see if the shared hand and player hand can make the chosen hand, and if so return that hand. This approach also requires substeps (how to decide if the hand can be formed, how to walk the possible hands, and so on) but sometimes (and in this case it is better) you can break it down faster.

The only way you get to choose between two approaches is if you are able to come up with both of them. A very good way to practice looking for multiple approaches to problems is to try to solve as many of the problems in the previous SRM using two different approaches. By doing this, you stretch your mind into looking for these different solutions, increasing your chances of finding the more elegant solution, or the faster one to type, or even the one that looks easier to debug.

Backtracking from a Flawed Approach

As demonstrated in the previous section, it is very possible for there to exist more than one way to plan an approach to a problem. It may even hit you in the middle of coding your approach how to more elegantly solve the problem. One of the hardest disciplines to develop while competing in topcoder Single Round Matches is the facility to stick to the approach you've chosen until you can prove without a shadow of a doubt that you made a mistake in the approach and the solution will not work. Remember that you are not awarded points for code elegance, or for cleverness, or for optimized code. You are granted points solely on the ability to quickly post correct code. If you come up with a more elegant solution than the one you're in the middle of typing up, you have to make the split second analysis of how much time you'll lose for changing your current approach, and in most cases it isn't worth it.

There is no easy answer to planning the right approach the first time. If you code up a solution and you know it is right but has bugs this is much easier to repair than the sudden realization that you just went the entirely wrong direction. If you get caught in this situation, whatever you do, don't erase your code! Relabel your main function and any subfunctions or data structures that may be affected by further changes. The

reason is because while you may desire a clean slate, you must accept that some of your previous routines may be the same, and retracing your steps by retyping the same code can be counterproductive to your thinking anew. Furthermore, by keeping the old code you can test against it later looking for cases that will successfully challenge other coders using the same flawed approach.

Conclusion

Planning an approach is not a science, although there is a lot of rigor in the thought involved. Rather, it is mainly educated guesswork coupled with successful planning. By being creative, economical, and thorough about your thought process you can become more successful and confident in your solutions and the time you spend thinking the problem through will save you time later in the coding and debugging. This ability to plan your code before the fingers hit the keys only develops through lots of practice, but this diligence is rewarded with an increasing ability to solve problems and eventually a sustained rating increase.

Mentioned in this writeup:

TCI '02 Round 2 Div I Med – MatArith
SRM 170 Div I Med – CityLink
SRM 172 Div I Med – Fifteen
SRM 206 Div I Hard – HexagonIntersections
SRM 195 Div I Easy – FanFailure
SRM 167 Div I Med – TeamPhoto
SRM 174 Div I Easy – BirthdayOdds
SRM 191 Div I Hard – MagicianTour
SRM 210 Div II Hard – TopographicalImage
SRM 206 Div I Easy – OmahaLow