Session 2: Problem Solving Strategies

2.1 Introduction

Problem Solving can be defined as the *ability* to take a given *problem description* and write an *original program* to *solve it*.

> From the definition problem solving may sound very natural and easy; contrary to that problem solving is not easy. It is true a few individuals make it look easy – the "naturals," the programming world's equivalent of a gifted athlete, like Uasin Bolt. For those few individuals, high – level ideas are effortlessly translated into source code. To make a *Java metaphor*, it is as if their brains execute Java natively, while the rest of us have to run a virtual machine, interpreting as we go. In as much as it is not easy and not very natural, it is not fatal becoming a programmer – otherwise, the world would have very few programmers.

2.2 Why is learning to solve programming problems not so easy?

In part, it is because problem solving is a different activity from learning programming syntax and therefore uses a different set of mental "muscles". Learning programming syntax, reading programs, memorizing elements of an application programming interface – these are mostly analytical "left brain" activities. Writing an original program using previously learned tools and skills is a creative "right brain" activity.

Suppose you want to cook "ugali" to take for lunch for the first time in your life (assuming you have the ingredients). It is not very easy, isn't it – though it looks very easy and natural when your dad or mum cooks the same ugali. You need to locate all the ingredients, put water in the cooking pot. Light a fire, then put the cooking pot which has water onto the fire; leave to boil and start cooking ugali (obvious there is also another process). The ultimate is to have ugali cooked in your own way. That is problem solving, and it's a creative activity. Believe it or not, when you design an original problem, your mental process is quite similar to that of the person who if figuring out how to cook ugali for the first time and quite different from that of a person debugging an existing `for` loop.

Instead of learning by trial and error, you can learn problem solving in a systematic way. That is all what you will learn throughout this unit. You will learn techniques to organize your thoughts, procedures to discover solutions, and strategies to apply to certain classes of problems. By studying these approaches, you can unlock your creativity. Make no mistake: Programming, and especially problem solving, is a creative activity. Creativity is mysterious, and no one can say exactly how the creative mind functions. Yet, if we can learn music composition, take advice on creative writing, or much still be shown how to paint, then we can learn to creatively solve programming problems, too. In this unit, we will try help you develop you latent problem-solving abilities so that you will know what you should do when faced with a programming problem.

2.3 Strategies for Problem Solving

When people use the term in ordinary conversation, they often mean something very different from what we mean here. If your 1997 Honda Civic has blue smoke coming from the tailpipe, is idling roughly, and has fuel efficiency, this is a problem that can be solved with automotive knowledge, diagnosis, replacement equipment, and common shop tools. If you tell your friends about problem, though, one of them might say, "*Hey, you should trade that old Honda in for something new. Problem solved*." But your friend's suggestion would not really be a solution to the problem – it would be a way to avoid the problem.

Problems include constraints, unbreakable rules about the problem or the way in which the problem must be solved. With the broken-down Civic, one of the constraints is that you want to fix the current car, not purchase a new car. The constraints might also include the overall cost of the repairs, or a requirement that no new tools can be purchased just for this repair.

When solving a problem with a program, you also have constraints. Common constraints include the programming language, platform (does it run on a PC, or an iPhone, or what?), performance (a game program may require graphics to be updated at least 30 times a second, a business application might have a maximum time response to user input), or memory footprint. Sometimes the constraint involves what other code you can reference: Maybe the program cannot include certain open-source code, or maybe the opposite – maybe it can use only open source.

For programmers, then, we can define *problem solving* as *writing an original program that performs a particular set of tasks* and *meets all stated constraints*.

It is also important to note that, the problems you will face as programmer are solvable, but many programmers still resort to dubious approaches. In some cases, they do so accidentally. In other cases, the removal of constraints is deliberate, a ploy to meet a deadline imposed by a boss or an instructor. In still other cases, the programmer just does not know how to meet all the constraints. In the worst cases I have seen, the programming student has paid someone else to write the program. Regardless of the motivations, we must always be diligent to avoid take shortcuts.

2.3.1 Classic Puzzle

As you progress, you will notice that although the particulars of the source code change from one problem area to the next, certain patterns will emerge in the approaches we take. This is great news because this is what eventually allows us to confidently approach any problem, whether we have extensive experience in that problem are or not. Expert problem solvers are quick to recognize an

analogy, an exploitable similarity between a solved problem and an unsolved problem. If we recognize that a feature of a problem A is analogous to a feature of problem B and we have already solved problem B, we have a valuable insight into solving problem A.

In this section, we will discuss a classic problem from outside the world of programming that have lessons we can apply to programming problems.

The Fox, the Goose, and the Corn

This is a classic problem, it is a riddle about a farmer who needs to cross a river. You have probably encountered it previously in one form or another.

PROBLEM: HOW TO CROSS THE RIVER?

A farmer with a fox, a goose, and a sack of corn needs to cross a river. The farmer has a towboat, but there is room for only the farmer and one of his three items. Unfortunately, both the fox and the goose are hungry. The fox cannot be left alone with the goose, or the fox will eat the goose. Likewise the goose cannot be left alone with sack of corn, or the goose will eat the corn. How does the farmer get everything across the river?

The setup for this problem is shown Figure 2-1. If you have never encountered this problem before, stop here and spend a few minutes trying to solve it. If you have heard this riddle before, try to remember the solution and whether you were able to solve the riddle on your own.

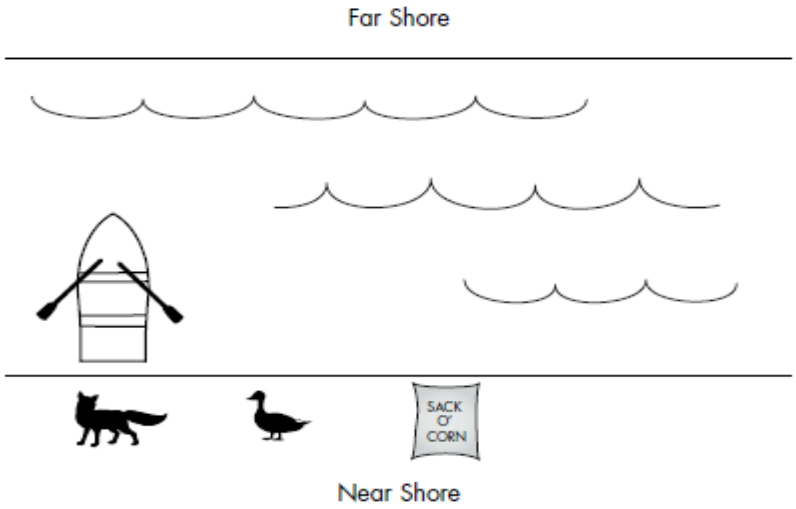

Figure 2-1: The fox, the goose, and the sack of corn. The boat can carry one item at a time.

Few people are able to solve this riddle, at least without a hint. I know I wasn't. Here's how the reasoning usually goes. Since the farmer can take only one thing at a time, he'll need multiple trips

to take everything to the far shore. On the first trip, if the farmer takes the fox, the goose would be left with the sack of corn, and the goose would eat the corn. Likewise, if the farmer took the sack of corn on the first trip, the fox would be left with the goose, and the fox would eat the goose. Therefore, the farmer must take the goose on the first trip, resulting in the configuration shown in Figure 2-2.

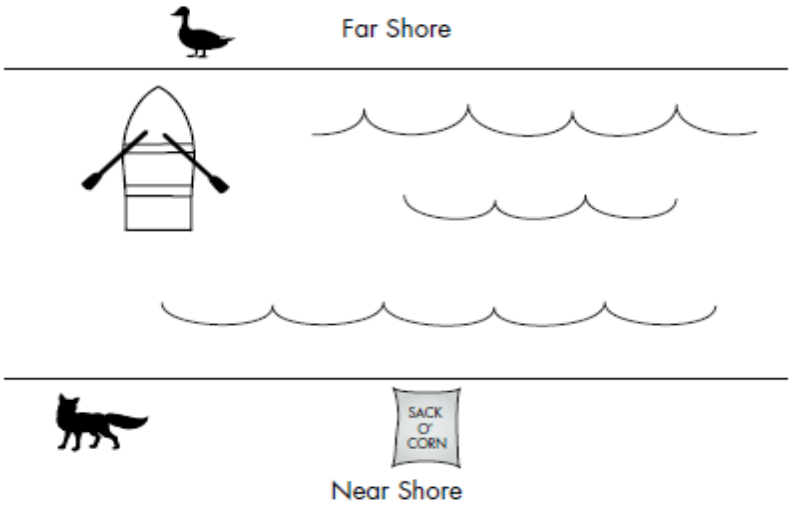

Figure 2-2: The required first step for solving the problem of the fox, the goose, and the sack of corn. From this step, however, all further steps appear to end in failure.

So far, so good. But on the second trip, the farmer must take the fox or the corn. Whatever the farmer takes, however, must be left on the far shore with the goose while the farmer returns to the near shore for the remaining item. This means that either the fox and goose will be left together or the goose and corn will be left together. Because neither of these situations is acceptable, the problem appears unsolvable.

Again, if you have seen this problem before, you probably remember the key element of the solution. The farmer has to take the goose on the first trip, as explained before. On the second trip, let's suppose the farmer takes the fox. Instead of leaving the fox with the goose, though, the farmer *takes the goose back* to the near shore. Then the farmer takes the sack of corn across, leaving the fox and the corn on the far shore, while returning for a fourth trip with the goose. The complete solution is shown in Figure 2-3.

This puzzle is difficult because most people never consider taking one of the items back from the far shore to the near shore. Some people will even suggest that the problem is unfair, saying something like, "You didn't say I could take something back!" This is true, but it's also true that nothing in the problem description suggests that taking something back is prohibited.

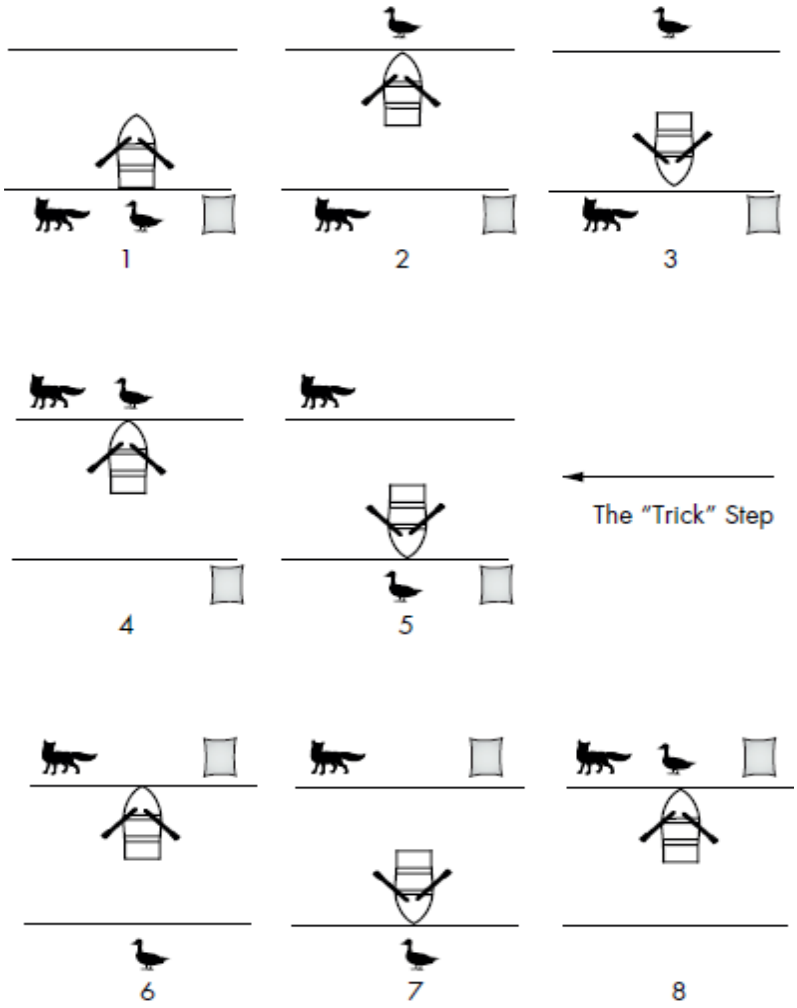

Figure 2-3: Step-by-step solution to the fox, goose, and corn puzzle

Think about how much easier the puzzle would be to solve if the possibility of taking one of the items back to the near shore was made explicit: *The farmer has a rowboat that can be used to transfer items in either direction, but there is room only for the farmer and one of his three items*. With that suggestion in plain sight, more people would figure out the problem. This illustrates an important principle of problem solving: If you are unaware of all possible actions you could take, you may be unable to solve the problem. We can refer to these actions as operations. By enumerating all the possible operations, we can solve many problems by testing every combination of operations until we find one that works. More generally, by restating a problem in more formal terms, we can often uncover solutions that would have otherwise eluded us.

Let's forget that we already know the solution and try stating this particular puzzle more formally. First, we'll list our constraints. The key constraints here are:

1. The farmer can take only one item at a time in the boat.

2. The fox and goose cannot be left alone on the same shore.
3. The goose and corn cannot be left alone on the same shore.

This problem is a good example of the importance of constraints. If we remove any of these constraints, the puzzle is easy. If we remove the first constraint, we can simply take all three items across in one trip. Even if we can take only two items in the boat, we can take the fox and corn across and then go back for the goose. If we remove the second constraint (but leave the other constraints in place), we just have to be careful, taking the goose across first, then the fox, and finally the corn. Therefore, if we forget or ignore any of the constraints, we will end up with a solution that does solve the problem.

Next, let's list the operations. There are various ways of stating the operations for this puzzle. We could make a specific list of the actions we think we can take:

1. Operation: Carry the fox to the far side of the river.
2. Operation: Carry the goose to the far side of the river.
3. Operation: Carry the corn to the far side of the river.

Remember, though, that the goal of formally restating the problem is to gain insight for a solution. Unless we have already solved the problem and discovered the "hidden" possible operation, taking the goose back to the near side of the river, we're not going to discover it in making our list of actions. Instead, we should try to make operations generic, or parameterized.

1. Operation: Row the boat from one shore to the other.
2. Operation: If the boat is empty, load an item from the shore.
3. Operation: If the boat is not empty, unload the item to the shore.

By thinking about the problem in the most general terms, this second list of operations will allow us to solve the problem without the need for an "ah-hah!" moment regarding the trip back to the near shore with the goose. If we generate all possible sequences of moves, ending each sequence once it violates one of our constraints or reaches a configuration we've seen before, we will eventually hit upon the sequence of Figure 2-3 and solve the puzzle. The inherent difficulty of the puzzle will have been sidestepped through the formal restatement of constraints and operations.

Lessons Learned

What can we learn from the fox, the goose, and the corn?

Restating the problem in a more formal manner is a great technique for gaining insight into a problem. Many programmers seek out other programmers to discuss a problem, not just because other programmers may have *the answer* but also because *articulating the problem out loud* often *triggers new and useful thoughts*. **Restating** a **problem** is like having that discussion with another programmer, except that you are playing both parts.

The broader lesson ***is that thinking about the problem*** may be as *productive, or in some cases more productive*, than thinking ***about the solution***. In many cases, the *correct approach to the solution* is the solution.

2.4 General Problem-Solving Techniques

The example we have just discussed demonstrates some of the key techniques that are employed in problem solving. As we move on in this unit, we will look at specific programming problems and figure out ways to solve them, but first we need a general set of techniques and principles. Some problem areas have specific techniques, as we will see, but the rules below apply to almost any situation. If you make these a regular part of your problem-solving approach, you will always have a method to attack a problem.

2.4.1 Always Have a Plan

This is perhaps the most important rule. You must always have a plan, rather than engaging in directionless activity.

By this point, you should understand that having a plan is always possible. It's true that if you haven't already solved the problem in your head, then you can't have a plan for implementing a solution in code. That will come later. Even at the beginning, though, you should have a plan for how you are going to find the solution.

To be fair, the plan may require alteration somewhere along the journey, or you may have to abandon your original plan and concoct another. Why, then, is this rule so important? General Dwight D. Eisenhower was famous for saying, "I have always found that plans are useless, but planning is indispensable." He meant that battles are so chaotic that it is impossible to predict everything that could happen and have a predetermined response for every outcome. In that sense, then, plans are useless on the battlefield (another military leader, the Prussian Helmuth von Moltke, famously said that "no plan survives first contact with the enemy"). But no army can succeed without planning and organization. Through planning, a general learns what his army's capabilities are, how the different parts of the army work together, and so on.

In the same way, you must always have a plan for solving a problem. It may not survive first contact with the enemy—it may be discarded as soon as you start to type code into your source editor—but you must have a plan. Without a plan, you are simply hoping for a lucky break, the equivalent of the randomly typing monkey producing one of the plays of Shakespeare. Lucky breaks are uncommon, and those that occur may still require a plan. Many people have heard the

story of the discovery of penicillin: A researcher named Alexander Fleming forgot to close a petri dish one night and in the morning found that mold had inhibited the growth of the bacteria in the dish. But Fleming was not sitting around waiting for a lucky break; he had been experimenting in a thorough and controlled way and thus recognized the importance of what he saw in the petri dish. (If I found mold growing on something I left out the night before, this would not result in an important contribution to science.)

Planning also allows you to set intermediate goals and achieve them. Without a plan, you have only one goal: solve the whole problem. Until you have solved the problem, you won't feel you have accomplished anything. As you have probably experienced, many programs don't do anything useful until they are close to completion. Therefore, working only toward the primary goal inevitably leads to frustration, as there is no positive reinforcement from your efforts until the end. If instead, you create a plan with a series of minor goals, even if some seem tangential to the main problem, you will make measurable progress toward a solution and feel that your time has been spent usefully. At the end of each work session, you'll be able to check off items from your plan, gaining confidence that you will find a solution instead of growing increasingly frustrated.

2.4.2 Restate the Problem

As demonstrated especially by the fox, goose, and corn problem, restating a problem can produce valuable results. In some cases, a problem that looks very difficult may seem easy when stated in a different way or using different terms. Restating a problem is like circling the base of a hill that you must climb; before starting your climb, why not check out the hill from every angle to see whether there's an easier way up?

Restatement sometimes shows us the goal was not what we thought it was. I once read about a grandmother who was watching over her baby granddaughter while knitting. In order to get her knitting done, the grandmother put the baby next to her in a portable play pen, but the baby didn't like being in the pen and kept crying. The grandmother tried all sorts of toys to make the pen more fun for the baby, until she realized that keeping the baby in the pen was just a means to an end. The goal was for the grandmother to be able to knit in peace. The solution: Let the baby play happily on the carpet, while the grandmother knits inside the pen. Restatement can be a powerful technique, but many programmers will skip it because it does not directly involve writing code or even designing a solution. This is another reason why having a plan is essential. Without a plan, your only goal is to have working code, and restatement is taking time away from writing code. With a plan, you can put "formally restate the problem" as your first step; therefore, completing the restatement officially counts as progress.

Even if a restatement doesn't lead to any immediate insight, it can help in other ways. For example, if a problem has been assigned to you (by a supervisor or an instructor), you can take your

restatement to the person who assigned the problem and confirm your understanding. Also, restating the problem may be a necessary prerequisite step to using other common techniques, like reducing or dividing the problem.

More broadly, restatement can transform whole problem areas. The technique I employ for recursive solutions, is a method to restate recursive problems so that I can treat them the same as iterative problems.

2.4.3 Divide the Problem

Finding a way to divide a problem into steps or phases can make the problem much easier. If you can divide a problem into two pieces, you might think that each piece would be half as difficult to solve as the original whole, but usually, it's even easier than that.

Here's an analogy that will be familiar if you have already seen common sorting algorithms. Suppose you have 100 files you need to place in a box in alphabetical order, and your basic alphabetizing method is effectively what we call an insertion sort: You take one of the files at random, put it in the box, then put the next file in the box in the correct relationship to the first file, and then continue, always putting the new file in its correct position relative to the other files, so that at any given time, the files in the box are alphabetized. Suppose someone initially separates the files into 4 groups of roughly equal size, A–F, G–M, N–S, and T–Z, and tells you to alphabetize the 4 groups individually and then drop them one after the other into the box.

If each of the groups contained about 25 files, then one might think that alphabetizing 4 groups of 25 is about the same amount of work as alphabetizing a single group of 100. But it's actually far less work because the work involved in inserting a single file grows as the number of files already filed grows—you have to look at each file in the box to know where the new file should be placed. (If you doubt this, think of a more extreme version—compare the thought of ordering 50 groups of 2 files, which you could probably do in under a minute, with ordering a single group of 100 files.)

In the same way, dividing a problem can often lower the difficulty by an order of magnitude. Combining programming techniques is much trickier than using techniques alone. For example, a section of code that employs a series of if statements inside a while loop that is itself inside a for loop will be more difficult to write—and to read—than a section of code that employs all those same control statements sequentially.

2.4.4 Start with What You Know

First-time novelists are often given the advice "write what you know." This doesn't mean that novelists should try only to craft works around incidents and people they have directly observed in their own lives; if this were the case, we could never have fantasy novels, historical fiction, or many other popular genres. But it means that the further away a writer gets from his or her own experience, the more difficult writing may be.

In the same way, when programming, you should try to start with what you already know how to do and work outward from there. Once you have divided the problem up into pieces, for example, go ahead and complete any pieces you already know how to code. Having a working partial solution may spark ideas about the rest of the problem. Also, as you may have noticed, a common theme in problem solving is making useful progress to build confidence that you will ultimately complete the task. By starting with what you know, you build confidence and momentum toward the goal.

The "start with what you know" maxim also applies in cases where you haven't divided the problem. Imagine someone made a complete list of every skill in programming: writing a C++ class, sorting a list of numbers, finding the largest value in a linked list, and so on. At every point in your development as a programmer, there will be many skills on this list that you can do well, other skills you can use with effort, and then the other skills that you don't yet know. A particular problem may be entirely solvable with the skills you already have or it may not, but you should fully investigate the problem using the skills already in your head before looking elsewhere. If we think of programming skills as tools and a programming problem as a home repair project, you should try to make the repair using the tools already in your garage before heading to the hardware store.

This technique follows the principles we have already discussed. It follows a plan and gives order to our efforts. When we begin our investigation of a problem by applying the skills we already have, we may learn more about the problem and its ultimate solution.

2.4.5 Reduce the Problem

With this technique, when faced with a problem you are unable to solve, you reduce the scope of the problem, by either adding or removing constraints, to produce a problem that you do know how to solve. We will see this technique in action later, but here is a basic example. Suppose you are given a series of coordinates in three-dimensional space, and you must find the coordinates that are closest to each other. If you do not immediately know how to solve this, there are different ways you could reduce the problem to seek a solution. For example, what if the coordinates are in two-dimensional space, instead of three-dimensional space? If that does not help, what if the points

lie along a single line so that the coordinates are just individual numbers (C++ doubles, let's say)? Now the question essentially becomes, in a list of numbers, find the two numbers with the minimum absolute difference.

Or you could reduce the problem by keeping the coordinates in three dimensional space but have only three values, instead of an arbitrary-sized series. So instead of an algorithm to find the smallest distance between any two coordinates, it's just a question of comparing coordinate A to coordinate B, then B to C, and then A to C.

These reductions simplify the problem in different ways. The first reduction eliminates the need to compute the distance between three-dimensional points. Maybe we don't know how to do that yet, but until we figure that out, we can still make progress toward a solution. The second reduction, by contrast, focuses almost entirely on computing the distance between three dimensional points but eliminates the problem of finding a minimal value in an arbitrary-sized series of values.

Of course, to solve the original problem, we will eventually need the skills involved in both reductions. Even so, reduction allows us to work on a simpler problem even when we can't find a way to divide the problem into steps. We know we're not working on the full problem, but the reduced problem has enough in common with the full problem that we will make progress toward the ultimate solution. Many times, programmers discover they have all the individual skills necessary to solve the problem, and by writing code to solve each individual aspect of the problem, they see how to combine the various pieces of code into a unified whole.

Reducing the problem also allows us to pinpoint exactly where the remaining difficulty lies. Beginning programmers often need to seek out experienced programmers for assistance, but this can be a frustrating experience for everyone involved if the struggling programmer is unable to accurately describe the help that is needed. One never wants to be reduced to saying, "Here is my program, and it doesn't work. Why not?" Using the problem-reduction technique, one can pinpoint the help needed, saying something like, "Here is some code I wrote. As you can see, I know how to find the distance between two three-dimensional coordinates, and I know how to check whether one distance is less than another. But I cannot seem to find a general solution for finding the pair of coordinates with the minimum distance."

2.4.6 Look for Analogies

An analogy, for our purposes, is a similarity between a current problem and a problem already solved that can be exploited to help solve the current problem. The similarity may take many

forms. Sometimes it means the two problems are really the same problem. This is the situation we had with the fox, goose, and corn problem.

Most analogies are not that direct. Sometimes the similarity concerns only part of the problems. For example, two number-processing problems might be different in all aspects except that both of them work with numbers requiring more precision than that given by built-in floating point data types; you won't be able to use this analogy to solve the whole problem, but if you've already figured out a way to handle the extra precision issue, you can handle that same issue the same way again.

Although recognizing analogies is the most important way you will improve your speed and skill at problem solving, it is also the most difficult skill to develop. The reason it is so difficult at first is that you can't look for analogies until you have a storehouse of previous solutions to reference.

This is where developing programmers often try to take a shortcut, finding code that is similar to the needed code and modifying from there. For several reasons, though, this is a mistake. First, if you do not complete a solution yourself, you will not have fully understood and internalized it. Put simply, it is very difficult to correctly modify a program that you do not fully understand. You don't need to have written code to fully understand, but if you could not have written the code, your understanding will be necessarily limited. Second, every successful program you write is more than a solution to a current problem; it is a potential source of analogies to solve future problems. The more you rely on other programmers' code now, the more you will have to rely on it in the future.

2.4.7 Experiment

Sometimes the best way to make progress is to try things and observe the results. Note that experimentation is not the same as guessing. When you guess, you type some code and hope that it works, having no strong belief that it will. An experiment is a controlled process. You hypothesize what will happen when certain code is executed, try it out, and see whether your hypothesis is correct. From these observations, you gain information that will help you solve the original problem.

Experimentation may be especially helpful when dealing with application programming interfaces or class libraries. Suppose you are writing a program that uses a library class representing a vector (in this context, a one-dimensional array that automatically grows as more items are added), but you've never used this vector class before, and you're not sure what happens when an item is deleted from the vector. Instead of forging ahead with solving the original problem while uncertainties swirl inside your head, you could create a short, separate program just to play around

with the vector class and to specifically try out the situations that concern you. If you spend a little time on the "vector demonstrator" program, it might become a reference for future work with the class.

Other forms of experimentation are similar to debugging. Suppose a certain program is producing output that is backward from expectations—for example, if the output is numerical, the numbers are as expected, but in the reverse order. If you do not see why this is occurring after reviewing your code, as an experiment, you might try modifying the code to deliberately make the output backward (run a loop in the reverse direction, perhaps). The resulting change, or lack of change, in the output may reveal the problem in your original source code or may reveal a gap in your understanding. Either way, you are closer to a solution.

2.4.8 Do not Get Frustrated

The final technique is not so much a technique, but a maxim: Do not get frustrated. When you are frustrated, you will not think as clearly, you will not work as efficiently, and everything will take longer and seem harder. Even worse, frustration tends to feed on itself, so that what begins as mild irritation ends as outright anger.

When I give this advice to new programmers, they often retort that while they agree with my point in principle, they have no control over their frustrations. Is not asking a programmer not to get frustrated at lack of success like asking a little boy not to yell out if he steps on a tack? The answer is no. When someone steps on a tack, a strong signal is immediately sent through the central nervous system, where the lower depths of the brain respond. Unless you know you are about to step on the tack, it is impossible to react in time to countermand the automatic response from the brain. So we will let the little boy off the hook for yelling out.

The programmer is not in the same boat. At the risk of sounding like a self-help guru, a frustrated programmer isn't responding to an external stimulus. The frustrated programmer is not angry with the source code on the monitor, although the programmer may express the frustration in those terms. Instead, the frustrated programmer is angry at himself or herself. The source of the frustration is also the destination, the programmer's mind.

When you allow yourself to get frustrated—and I use the word "allow" deliberately—you are, in effect, giving yourself an excuse to continue to fail. Suppose you're working on a difficult problem and you feel your frustration rise. Hours later, you look back at an afternoon of gritted teeth and pencils snapped in anger and tell yourself that you would have made real progress if you had been able to calm down. In truth, you may have decided that giving in to your anger was easier than facing the difficult problem.

Ultimately, then, avoiding frustration is a decision you must make. However, there are some thoughts you can employ that will help. First of all, never forget the first rule, that you should always have a plan, and that while writing code that solves the original problem is the goal of that plan, it is not the only step of that plan. Thus, if you have a plan and you are following it, then you are making progress and you must believe this. If you have run through all the steps on your original plan and you are still not ready to start coding, then it is time to make another plan.

Also, when it comes down to getting frustrated or taking a break, you should take a break. One trick is to have more than one problem to work on so that if this one problem has you stymied, you can turn your efforts elsewhere. Note that if you successfully divide the problem, you can use this technique on a single problem; just block out the part of the problem that has you stuck, and work on something else. If you do not have another problem you can tackle, get out of your chair and do something else, something that keeps your blood flowing but does not make your brain hurt: Take a walk, do the laundry, go through your stretching routine (*if you are signing up to be a programmer, sitting at a computer all day, I highly recommend developing a stretching routine!*). ***Do not think about the problem until your break is over***.