

1

STRATEGIES FOR PROBLEM SOLVING



This book is about problem solving, but what is problem solving, exactly? 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 lost 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 your 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 wouldn’t 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, how long the repair will take, 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 can't 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.

Beginning programmers are often so eager to accomplish the first part of that definition—writing a program to perform a certain task—that they fail on the second part of the definition, meeting the stated constraints. I call a program like that, one that appears to produce correct results but breaks one or more of the stated rules, a *Kobayashi Maru*. If that name is unfamiliar to you, it means you are insufficiently familiar with one of the touchstones of geek culture, the film *Star Trek II: The Wrath of Khan*. The film contains a subplot about an exercise for aspiring officers at Starfleet Academy. The cadets are put aboard a simulated starship bridge and made to act as captain on a mission that involves an impossible choice. Innocent people will die on a wounded ship, the *Kobayashi Maru*, but to reach them requires starting a battle with the Klingons, a battle that can only end in the destruction of the captain's ship. The exercise is intended to test a cadet's courage under fire. There's no way to win, and all choices lead to bad outcomes. Toward the end of the film, we discover that Captain Kirk modified the simulation to make it actually winnable. Kirk was clever, but he did not solve the dilemma of the *Kobayashi Maru*; he avoided it.

Fortunately, the problems you will face as a programmer are solvable, but many programmers still resort to Kirk's approach. In some cases, they do so accidentally. ("Oh, shoot! My solution only works if there are a hundred data items or fewer. It's supposed to work for an unlimited data set. I'll have to rethink this.") 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 doesn't know how to meet all of 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 the *Kobayashi Maru*.

Classic Puzzles

As you progress through this book, 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 area 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 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'll discuss classic problems from outside the world of programming that have lessons we can apply to programming problems.

The Fox, the Goose, and the Corn

The first classic problem we will discuss 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 rowboat, 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 the 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 in Figure 1-1. 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.

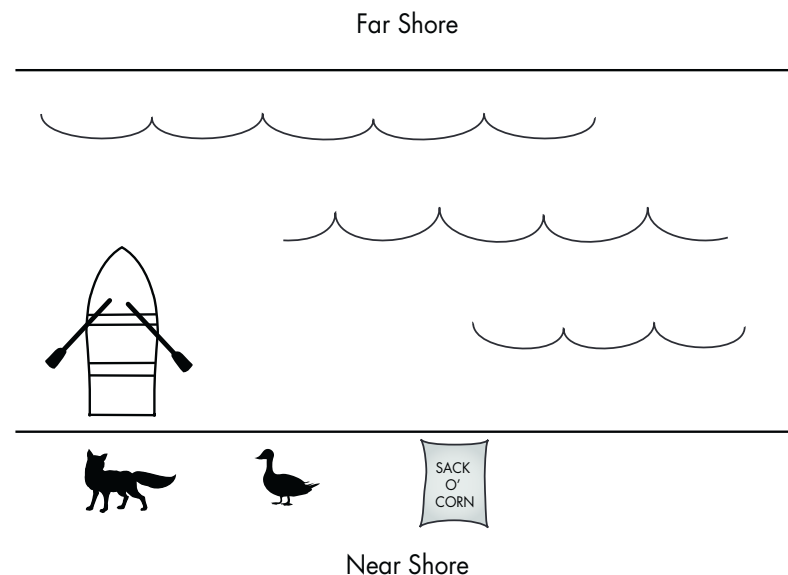


Figure 1-1: The fox, the goose, and the sack of corn. The boat can carry one item at a time. The fox cannot be left on the same shore as the goose, and the goose cannot be left on the same shore as the sack of corn.

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 1-2.

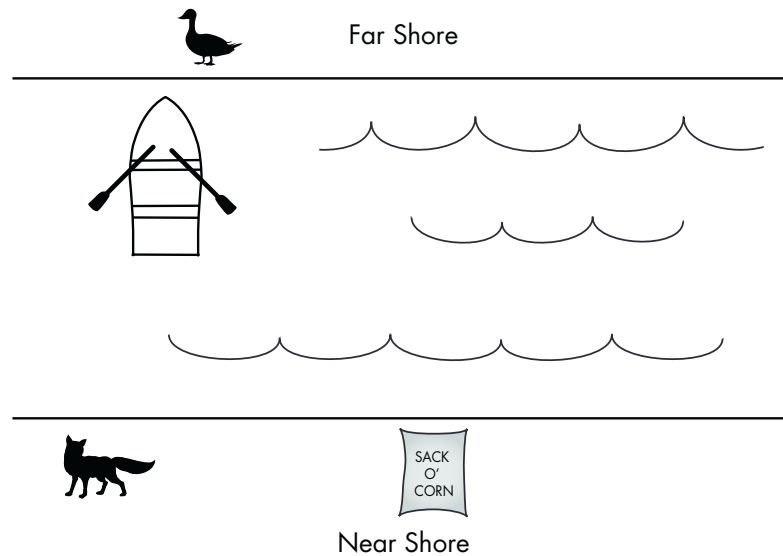


Figure 1-2: The required first step for solving the problem of the fox, the goose, and the sack of corn. From this step, however, all further steps appear to end in failure.

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 1-3.

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.

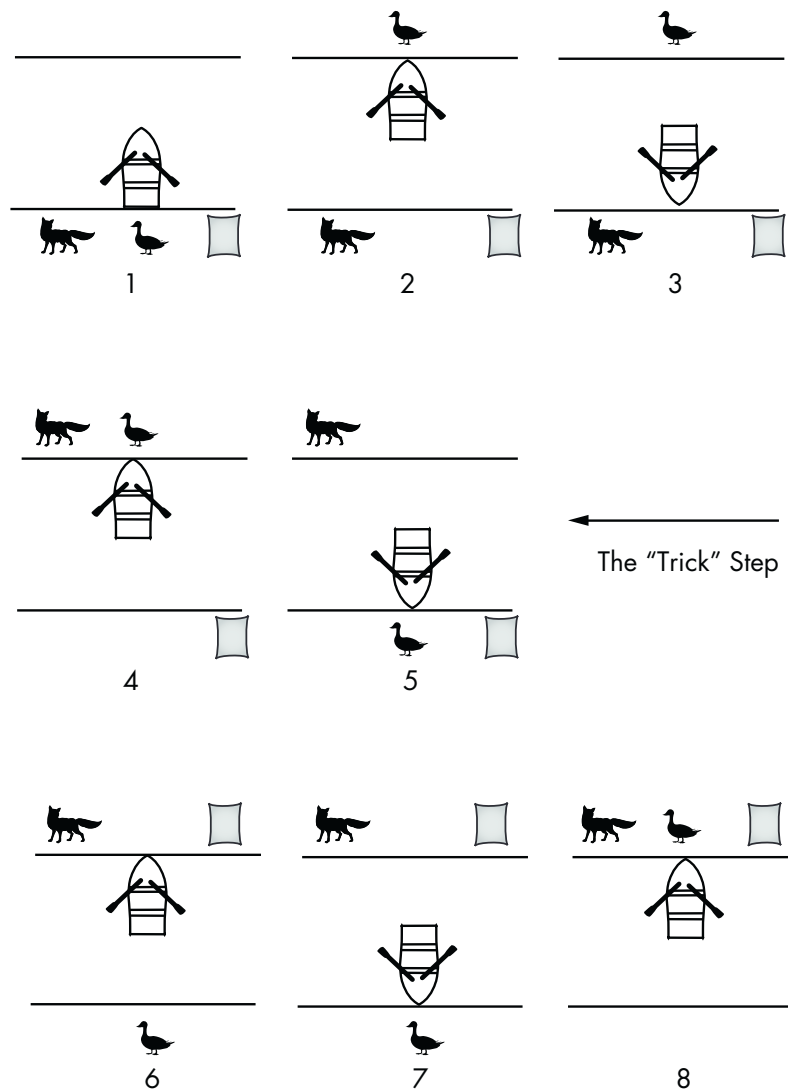


Figure 1-3: Step-by-step solution to the fox, goose, and corn puzzle

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 Kobayashi Maru.

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 1-3 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.

Sliding Tile Puzzles

The sliding tile puzzle comes in different sizes, which, as we'll see later, offers a particular solving mechanism. The following description is for a 3×3 version of the puzzle.

PROBLEM: THE SLIDING EIGHT

A 3×3 grid is filled with eight tiles, numbered 1 through 8, and one empty space. Initially, the grid is in a jumbled configuration. A tile can be slid into an adjacent empty space, leaving the tile's previous location empty. The goal is to slide the tiles to place the grid in an ordered configuration, from tile 1 in the upper left.

The goal of this problem is shown in Figure 1-4. If you've never tried a puzzle like this before, take the time to do so now. Plenty of sliding puzzle simulators can be found on the Web, but for our purposes it's better if you use playing cards or index cards to make your own game on a tabletop. A suggested starting configuration is shown in Figure 1-5.

1	2	3
4	5	6
7	8	

Figure 1-4: The goal configuration in the eight-tile version of the sliding tile puzzle. The empty square represents the empty space into which an adjacent tile may slide.

4	7	2
8	6	1
3	5	

Figure 1-5: A particular starting configuration for the sliding tile puzzle

This puzzle is quite different from the farmer with his fox, goose, and corn. The difficulty in that problem came from overlooking one of the possible operations. In this problem, that doesn't happen. From any given configuration, up to four tiles may be adjacent to the empty space, and any of those tiles can be slid into the empty space. That fully enumerates all possible operations.

The difficulty in this problem arises instead from the long chain of operations required by the solution. A series of sliding operations may move some tiles to their correct final positions while moving other tiles out of position, or it may move some tiles closer to their correct positions while moving others farther away. Because of this, it's difficult to tell whether any particular operation would make progress toward the ultimate goal. Without being able to measure progress, it's difficult to formulate a strategy. Many people who attempt a sliding tile puzzle simply move the tiles around randomly, hoping to hit upon a configuration from which a path to the goal configuration can be seen.

Nevertheless, there are strategies for sliding tile puzzles. To illustrate one approach, let's consider the puzzle for a smaller grid that is rectangular but not square.

PROBLEM: THE SLIDING FIVE

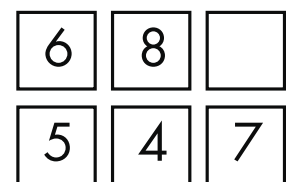
A 2x3 grid is filled with five tiles, numbered 4 through 8, and one empty space. Initially, the grid is in a jumbled configuration. A tile can be slid into an adjacent empty space, leaving the tile's previous location empty. The goal is to slide the tiles to place the grid in an ordered configuration, from tile 4 in the upper left.

You may have noticed that our five tiles are numbered 4 through 8 instead of 1 through 5. The reason for this will become clear shortly.

Although this is the same basic problem as the sliding eight, it is much easier with only five tiles. Try the configuration shown in Figure 1-6.

If you play around with these tiles for just a few minutes, you will probably hit upon a solution. From playing around with small-count tile puzzles, I have developed a particular skill. It is this one skill, coupled with an observation we will discuss shortly, that I use to solve all sliding tile puzzles.

I call my technique *the train*. It's based on the observation that a circuit of tile positions that includes the empty space forms a train of tiles that can be rotated anywhere along the circuit while preserving the relative ordering of the tiles. Figure 1-7 illustrates the smallest possible train of four positions. From the first configuration, the 1 can slide into the empty square, the 2 can slide into the space vacated by the 1, and finally the 3 can slide into the space vacated by the 2. This leaves the empty space adjacent to the 1, which allows the train to continue and, thus, the tiles to be effectively rotated anywhere along the train path.



6	8	
5	4	7

Figure 1-6: A particular starting configuration for a reduced, 2x3 sliding tile puzzle

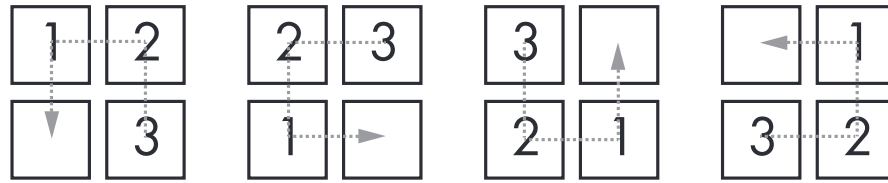


Figure 1-7: A “train,” a path of tiles that begins adjacent to the empty square and can slide like a train of cars through the puzzle

Using a train, we can move a series of tiles while maintaining their relative relationship. Now let’s return to the previous 2×3 grid configuration. Although none of the tiles in this grid is in its correct final position, some tiles are adjacent to the tiles they need to border in the final configuration. For example, in the final configuration, the 4 will be above the 7, and currently those tiles are adjacent. As shown in Figure 1-8, we can use a six-position train to bring the 4 and 7 to their correct final positions. When we do that, the remaining tiles are nearly correct; we just need to slide the 8 over.

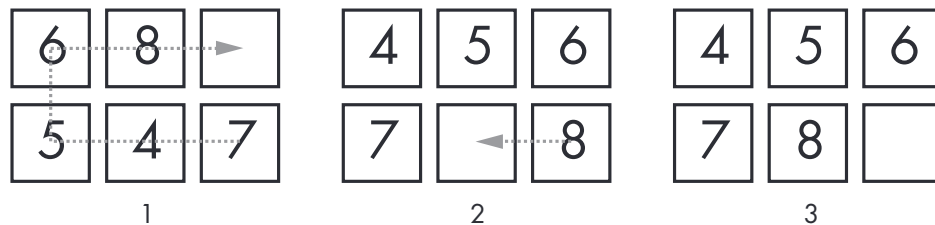


Figure 1-8: From configuration 1, two rotations along the outlined “train” bring us to configuration 2. From there, a single tile slide results in the goal, configuration 3.

So how does this one technique allow us to solve any sliding tile puzzle? Consider our original 3×3 configuration. We can use a six-position train to move the adjacent 1 and 2 tiles so that the 2 and 3 are adjacent, as shown in Figure 1-9.

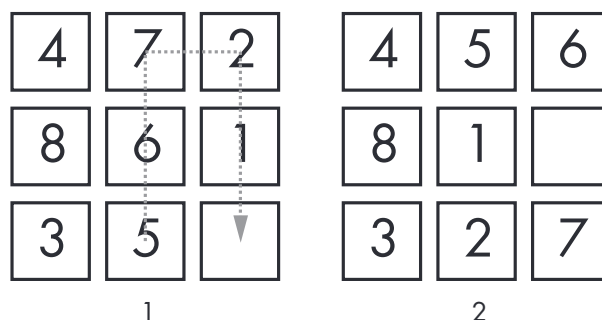


Figure 1-9: From configuration 1, tiles are rotated along the outlined “train” to reach configuration 2.

This puts 1, 2, and 3 in adjacent squares. With an eight-position train, we can shift the 1, 2, and 3 tiles to their correct final positions, as shown in Figure 1-10.

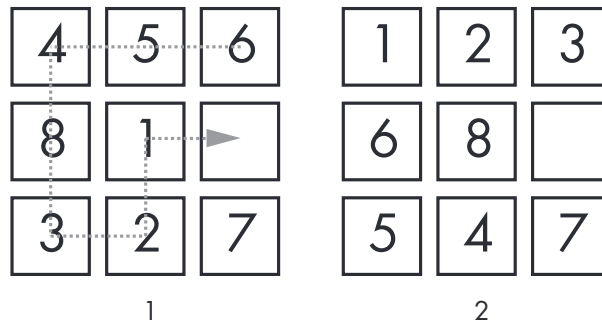


Figure 1-10: From configuration 1, tiles are rotated to reach configuration 2, in which tiles 1, 2, and 3 are in their correct final positions.

Notice the positions of tiles 4–8. The tiles are in the configuration I gave for the 2×3 grid. This is the key observation. Having placed tiles 1–3 in their correct positions, we can solve the rest of the grid as a separate, smaller, and easier puzzle. Note that we have to solve an entire row or column for this method to work; if we put tiles 1 and 2 in the correct positions but tile 3 is still out of place, there is no way to move something into the upper-right corner without moving one or both of the other upper-row tiles out of place.

This same technique can be used to solve even larger sliding tile puzzles. The largest common size is a 15-tile puzzle, a 4×4 grid. This can be solved piecemeal by first moving tiles 1–4 to their correct position, leaving a 3×4 grid, and then moving the tiles of the leftmost column, leaving a 3×3 grid. At that point, the problem has been reduced to an 8-tile puzzle.

Lessons Learned

What lessons can we learn from the sliding tile puzzles?

The number of tile movements is large enough that it is difficult or impossible to plan out a complete solution for a sliding tile puzzle from the initial configuration. However, our inability to plan a complete solution does not prevent us from making strategies or employing techniques to systematically solve the puzzle. In solving programming problems, we are sometimes faced with situations where we can't see a clear path to code the solution, but we must never allow this to be an excuse to forgo planning and systematic approaches. It's better to develop a strategy than to attack the problem through trial and error.

I developed my “train” technique from fiddling around with a small puzzle. Often, I use a similar technique in programming. When faced with an onerous problem, I experiment with a reduced version of the problem. These experiments frequently produce valuable insights.

The other lesson is that sometimes problems are divisible in ways that are not immediately obvious. Because moving a tile affects not only that tile but also the possible moves that can be made next, one might think that a sliding tile puzzle must be solved all in one step, not in stages. Looking for a way to divide a problem is usually time well spent. Even if you are unable to find a

clean division, you may learn something about the problem that helps you to solve it. When solving problems, working with a specific goal in mind is always better than random effort, whether you achieve that specific goal or not.

Sudoku

The sudoku game has become enormously popular through appearances in newspapers and magazines and also as a web-based and phone-based game. Variations exist, but we will briefly discuss the traditional version.

PROBLEM: COMPLETING A SUDOKU SQUARE

A 9×9 grid is partially filled with single digits (from 1–9), and the player must fill in the empty squares while meeting certain constraints: In each row and column, each digit must appear exactly once, and further, in each marked 3×3 area, each digit must appear exactly once.

If you have played this game before, you probably already have a set of strategies for completing a square in the minimum time. Let's focus on the key starting strategy by looking at the sample square shown in Figure 1-11.

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

Figure 1-11: An easy sudoku square puzzle

Sudoku puzzles vary in difficulty, their difficulty determined by the number of squares left to be filled. By this measure, this is a very easy puzzle. As 36 squares are already numbered, there are just 45 that must be filled to complete the puzzle. The question is, which squares should we attempt to fill in first?

Remember the puzzle constraints. Each of the nine digits must appear once in every row, in every column, and in every 3×3 area marked by the heavy borders. These rules dictate where we should begin our efforts. The 3×3 area in the middle of the puzzle already has numbers in eight of its nine squares. Therefore, the square in the very center can have only one possible value, the one value not already represented in another square in that 3×3 area. That's where we should start solving this puzzle. The missing number in that area is 7, so we would place that in the middle square.

With that value in place, note that the centermost column now has values in seven of its nine squares, which leaves only two squares remaining, each of which has to have a value not already in the column: The two missing numbers are 3 and 9. The constraint on this column would allow us to put either number in either place, but notice that 3 is already present in the third row and 9 is already present in the seventh row. Therefore, the row constraints dictate that 9 go in the third row of the middle column and 3 go in the seventh row of the middle column. These steps are summarized in Figure 1-12.

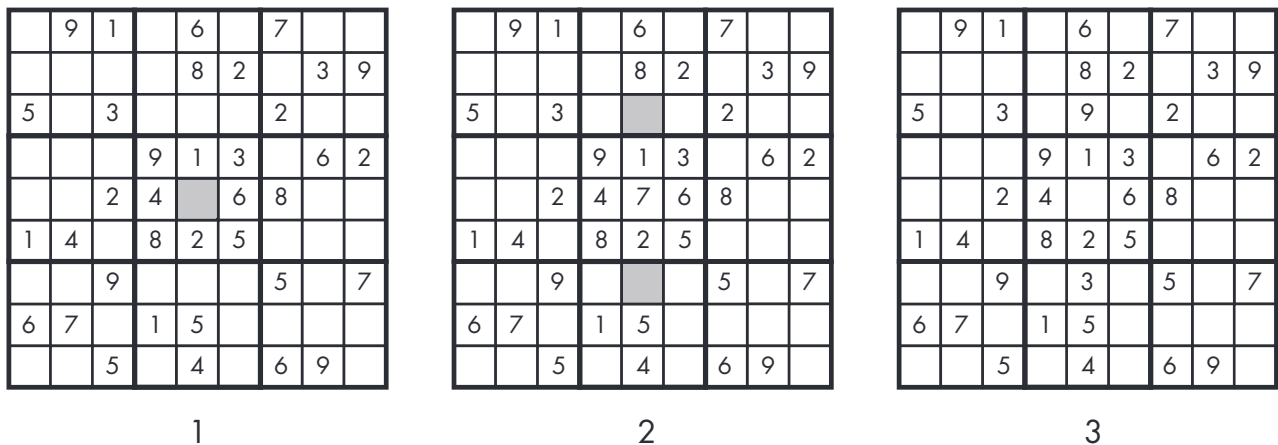


Figure 1-12: The first steps in solving the sample sudoku puzzle

We won't solve the entire puzzle here, but these first steps make the important point that we search for squares that have the lowest number of possible values—ideally, just one.

Lessons Learned

The main lesson from sudoku is that we should look for the most constrained part of the problem. While constraints are often what make a problem difficult to begin with (remember the fox, the goose, and the corn), they may also simplify our thinking about the solution because they eliminate choices.

Although we will not discuss artificial intelligence specifically in this book, there is a rule for solving certain types of problems in artificial intelligence called the “most constrained variable.” It means that in a problem where you are trying to assign different values to different variables to meet constraints, you should start with the variable that has the most constraints, or put another way, the variable that has the lowest number of possible values.

Here's an example of this sort of thinking. Suppose a group of coworkers wants to go to lunch together, and they've asked you to find a restaurant that everyone will like. The problem is, each of the coworkers imposes some kind of constraint on the group decision: Pam is a vegetarian, Todd doesn't like Chinese food, and so on. If your goal is to minimize the amount of time it takes to find a restaurant, you should start by talking to the coworker with the most onerous restrictions. If Bob has a number of broad food allergies, for example, it would make sense to start by finding a list of restaurants where he knows he can eat, rather than starting with Todd, whose dislike of Chinese food can be easily mitigated.

The same technique can often be applied to programming problems. If one part of the problem is heavily constrained, that's a great place to start because you can make progress without worrying that you are spending time on work that will later be undone. A related corollary is that you should start with the part that's obvious. If you can solve part of the problem, go ahead and do what you can. You may learn something from seeing your own code that will stimulate your imagination to solve the rest.

The Quarrasi Lock

You may have seen each of the previous puzzles before, but you should not have seen the last one in this chapter unless you have read this book previously, because I've made this one up myself. Read carefully because the wording of this problem is a little complicated.

PROBLEM: OPENING THE ALIEN LOCK

A hostile alien race, the Quarrasi, has landed on Earth, and you've been captured. You've managed to overpower your guards, even though they are enormous and tentacled, but to escape the (still grounded) spaceship, you have to open the massive door. The instructions for opening the door are, oddly enough, printed in English, but it's still no piece of cake. To open the door, you have to slide the three bar-shaped Kratzz along tracks that lead from the right receptor to the left receptor, which lies at the end of the door, 10 feet away.

That's easy enough, but you have to avoid setting off the alarms, which work as follows. On each Kratzz are one or more star-shaped crystal power gems known as Quinicrys. Each receptor has four sensors that light up if the number of Quinicrys in the column above is even. An alarm goes off if the number of lit sensors is ever exactly one. Note that each receptor's alarm is separate: You can't ever have exactly one sensor lit for the left receptor or for the right receptor. The good news is that each alarm is equipped with a suppressor, which keeps the alarm from sounding as long as the button is pressed. If you could press both suppressors at once, the problem would be easy, but you can't since you have short human arms rather than long Quarassi tentacles.

Given all of this, how do you slide the Kratzz to open the door without activating either alarm?

The starting configuration is shown in Figure 1-13, with all three Kratzz in the right receptor. For clarity, Figure 1-14 shows a bad idea: Sliding the uppermost Kratzz to the left receptor causes an alarm state in the right receptor. You might think that we could avoid the alarm with the suppressor, but remember that we just moved the upper Kratzz to the left receptor, so we're 10 feet away from the right receptor's suppressor.

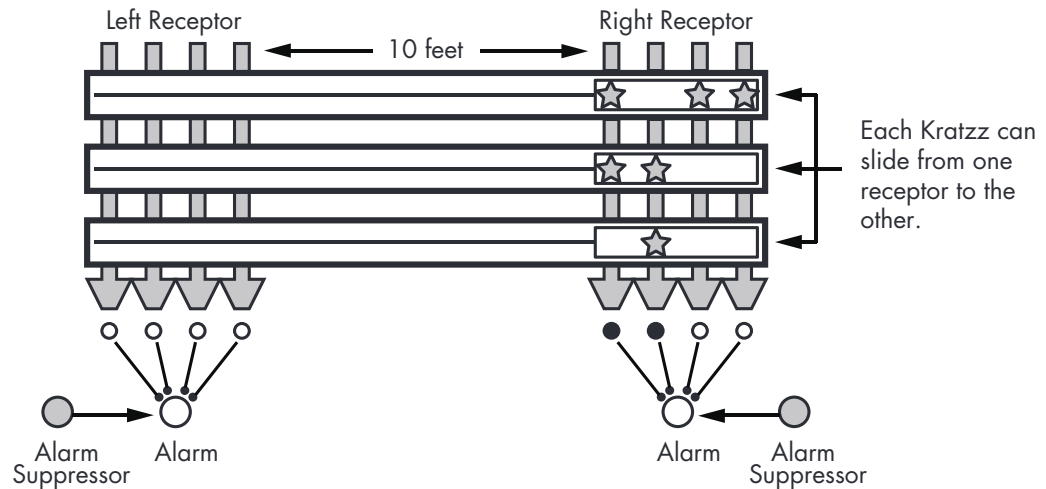


Figure 1-13: Starting configuration for the Quarrasi lock problem. You must slide the three Kratzz bars, currently in the right receptor, to the left receptor without setting off either alarm. A sensor is lit when an even number of star-shaped Quinicrys appear in the column above, and an alarm sounds if exactly one connected sensor lights up. Suppressors can keep an alarm from sounding, but only for the receptor where you are standing.

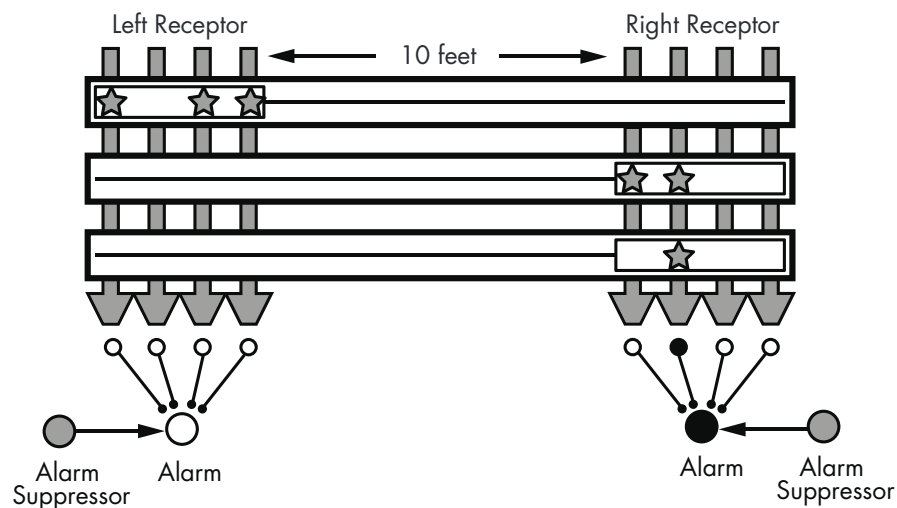


Figure 1-14: The Quarrasi lock in an alarm state. You just slid the upper Kratzz to the left receptor, so the right receptor is out of reach. The second sensor for the right alarm is lit because an even number of Quinicrys appears in the column above, and an alarm sounds when exactly one of its sensors is lit.

Before moving on, take some time to study this problem, and try to develop a solution. Depending on your point of view, this problem is not as hard as it looks. Seriously, think about it before moving on!

Have you thought about it? Were you able to come up with a solution? There are two possible paths to an answer here. The first path is trial and error: attempting various Kratzz moves in a methodical way and backing up to previous steps when you reach an alarm state until you find a series of moves that succeeds.

The second path is realizing that the puzzle is a trick. If you haven't seen the trick yet, here it is: This is just the fox, goose, and corn problem in an elaborate disguise. Although the rules for the alarm are written generally, there are only so many combinations for this specific lock. With only three Kratzz, we just have to know which combinations of Kratzz in a receptor are acceptable. If we label the three Kratzz *top*, *middle*, and *bottom*, then the combinations that create alarms are "top and middle" and "middle and bottom." If we rename *top* as *fox*, *middle* as *goose*, and *bottom* as *corn*, then the troublesome combinations are the same as in the other problem, "fox and goose" and "goose and corn."

This problem is therefore solved in the same way as the fox, goose, and corn problem. We slide the middle Kratzz (goose) over to the left receptacle. Then, we slide the top (fox) to the left, holding the left alarm's suppressor as we put the top (fox) into place. Next, we start sliding the middle (goose) back to the right receptacle. Then, we slide the bottom (corn) to the left, and finally, we slide the middle (goose) to the left once again, opening the lock.

Lessons Learned

The chief lesson here is the importance of recognizing analogies. Here, we can see that the Quarrasi lock problem is analogous to the fox, goose, and corn problem. If we discover that analogy early enough, we can avoid most of the work of the problem by translating our solution from the first problem rather than creating a new solution. Most analogies in problem solving won't be so direct, but they will happen with increasing frequency.

If you had trouble seeing the connection between this problem and the fox, goose, and corn problem, that's because I deliberately included as much extraneous detail as possible. The story that sets up the Quarrasi problem is irrelevant, as are the names for all of the alien technology, which serve to heighten the sense of unfamiliarity. Furthermore, the odd/even mechanism of the alarm makes the problem seem more complicated than it is. If you look at the actual positioning of the Quinicrys, you can see that the top Kratzz and the bottom Kratzz are opposites, so they don't interact in the alarm system. The middle Kratzz, however, interacts with the other two.

Again, if you didn't see the analogy, don't worry. You'll start to recognize them more after you put yourself on alert for them.

General Problem-Solving Techniques

The examples we have discussed demonstrate many of the key techniques that are employed in problem solving. In the rest of this book, we'll 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'll see, but the rules below apply to almost any situation. If you make these a regular part of your problem-solving approach, you'll always have a method to attack a problem.

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.

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 doesn't 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, which I share in a later chapter, is a method to restate recursive problems so that I can treat them the same as iterative problems.

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.

We'll discuss specific ways to divide problems in the chapters that follow, but you should always be alert to the possibility. Remember that some problems, like our sliding tile puzzle, often hide their potential subdivision. Sometimes the way to find a problem's divisions is to reduce the problem, as we'll discuss shortly.

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.

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'll see this technique in action in later chapters, but here's 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 don't 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 doesn't 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. In effect, it's like a deliberate, but temporary, Kobayashi Maru. 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’s my program, and it doesn’t work. Why not?” Using the problem-reduction technique, one can pinpoint the help needed, saying something like, “Here’s 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 can’t seem to find a general solution for finding the pair of coordinates with the minimum distance.”

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 and the Quarrasi lock 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 don’t complete a solution yourself, you won’t have fully understood and internalized it. Put simply, it’s very difficult to correctly modify a program that you don’t 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’s 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. We’ll talk in depth about “good reuse” and “bad reuse” in Chapter 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 don't 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're closer to a solution.

Don't Get Frustrated

The final technique isn't so much a technique, but a maxim: Don't get frustrated. When you are frustrated, you won't think as clearly, you won't 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. Isn't 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're about to step on the tack, it's impossible to react in time to countermand the automatic response from the brain. So we'll 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 isn't 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’re following it, then you are making progress and you must believe this. If you’ve run through all the steps on your original plan and you’re still not ready to start coding, then it’s 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 don’t have another problem you can tackle, get out of your chair and do something else, something that keeps your blood flowing but doesn’t make your brain hurt: Take a walk, do the laundry, go through your stretching routine (if you’re signing up to be a programmer, sitting at a computer all day, I highly recommend developing a stretching routine!). Don’t think about the problem until your break is over.

Exercises

Remember, to truly learn something you have to put it into practice, so work as many exercises as you can. In this first chapter, of course, we’re not yet discussing programming, but even so, I encourage you to try some exercises out. Think of these questions as warm-ups for your fingers before we start playing the real music.

- 1-1. Try a medium-difficulty sudoku puzzle (you can find these all over the Web and probably in your local newspaper), experimenting with different strategies and taking note of the results. Can you write a general plan for solving a sudoku?
- 1-2. Consider a sliding tile puzzle variant where the tiles are covered with a picture instead of numbers. How much does this increase the difficulty, and why?
- 1-3. Find a strategy for sliding tile puzzles different from mine.

- 1-4.** Search for old-fashioned puzzles of the fox, goose, and corn variety and try to solve them. Many of the great puzzles were originated or popularized by Sam Loyd, so you might search for his name. Furthermore, once you uncover (or give up and read) the solution, think of how you could make an easier version of the puzzle. What would you have to change? The constraints or just the wording?
- 1-5.** Try to write some explicit strategies for other traditional pencil-and-paper games, like crosswords. Where should you start? What should you do when you're stuck? Even simple newspaper games, like "Jumble," are useful for contemplating strategy.

