Chapter 5

Algorithmic Design

(This chapter is currently under heavy revision!)

All of our programs are composed of two parts:

1. Data.
2. Operations over that data.

We now turn our attention towards the these data structures and algorithms that make up our programs. Previously all

Of course, all but the simplest of toy programs contain many pieces of related data that we most organize in some way. We do so through various *containers* that efficiently hold data in various ways

Before we dive into the specifics of data structures and algorithms, however, we'll first take a step back and introduce some high-level concepts that will help structure our discussion for the rest of the course.

5.1 Data Organization

5.2 Algorithmic Design

Algorithm design is a daunting affair. You probably have experienced this first-hand on exams where you were asked to write a program to accomplish a task, *e.g.*, sorting the elements of an array. How do you go from a problem statement to a working algorithm? Much of the time, it feels like a game of chance; either you either "get it" or you don't with little room in between. And to be honest, this mindset has some merit. Algorithm design is a difficult task with a steep slope. In a heartbeat, you'll go from not knowing at all how to solve a problem to kicking yourself for not seeing the solution earlier. This is because most algorithms only require a single burst of inspiration, a spark of an insight, a key idea, *etc.*.

However, algorithm design is not entirely luck-based! There are ways we can be systematic about our design process so that we quickly and consistently hone in on a final solution. Here I want to touch on two complementary skills you will use in designing algorithms:

a Consulting Your Toolbox.

b Exploring the Problem Space.

5.3 Consulting Your Toolbox

One way of solving an algorithm is to know it already. This line of thinking is what motivates many books for "Cracking the Coding Questions" on internship or job interviews. If we know everything, then we'll never be stumped by an algorithms question!

Of course, there is clearly a problem with this line of reasoning. We can't possibly know the solution to every problem set before us beforehand. Rather than memorizing solutions to particular problems, we need to be able to generalize the solutions to problems that we have previously encountered so that we can apply those techniques to as many new problems as possible.

You can think of this process as building up an *algorithmic toolbox*, a collection of strategies and techniques that apply to large classes of problems. Because our computer programs consist of data and functions that manipulate that data, we can break up our toolbox into two parts:

a *Abstract data types* that organize our data so that we can efficiently access them as our program demands it, and *data structures* that provide concrete implementations of those abstract data types.

b *Programming patterns* that allow us to manipulate our data efficiently and in a readable manner.

5.3.1 Abstract Data Types

One dimension of your toolbox concerns how you organize your data. These various sorts of fundamental abstract data types and data structures are the focus of this course. An abstract data type defines (ADT) an *interface*, the allowable set of operations on data of a particular type. In contrast, a data structure provides a particular implementation for that abstract data type.

We organize our abstract data types broadly into four categories based on the relationships we maintain between the data:

- *Sequential types*, a linear ordering between data.
- *Hierarchical structures*, a parent-children relationship between data.
- *Mapping structures*, key-value relationship between data.
- *Graph structures*, arbitrary relationships between data.

For example, within the sequential types category, there are a number of abstract data types, including:

- Lists—Variable-size, homogeneous, sequential containers.
- Arrays—Fixed-size homogeneous, sequential, random-access containers.
- Tuples—Fixed-size heterogeneous, random-access containers.
- Queues—Homogeneous containers with first-in-last-out (FILO) behavior.
- Stacks—Homogeneous containers with first-in-first-out (FIFO) behavior.

Finally, each abstract data type has one or more data structures that implement that data type. In some cases, the implementing data structure is the same as the abstract data type. For example, an array defines an abstract data type but is also its own implementation. (In some contexts, you will see the term "Vector" used as the abstract data type that an array implements.) However, in most cases, there will be multiple data structures that implement a particular abstract data type. For example, the List abstract data type can be implemented by using a linked-based or array-based data structure.

When solving a problem, we must identify how we should organize our data. This in turn helps us identify which abstract data type we should use. Frequently, multiple abstract data types fulfill our needs, in which case, we may have to weigh trade-offs between them to choose one. Likewise, once we have identified an abstract data type, we will need to choose a particular implementing data structure, each with their own trade-offs, *e.g.*, performance between various operations or implementation complexity.

5.3.2 Programming Patterns

Once we've chosen how we represent our data, we must actually manipulate that data to obtain the desired result. We rarely invent all-new ways of manipulating data on the fly. Rather, we rely on number of *programming patterns* as a basis for our solution, and then refine it further to solve the particular problem at hand. These programming patterns are designed to accomplish some small task en route towards solving the overall problem. Examples of such patterns you have seen before:

- *Iteration*: using a for-loop to traverse an array.
- *Swapping*: swapping the contents of two variables.
- *Finger-pointing*: generalizing iteration across multiple containers by tracking indices into each container.
- *Link-walking*: maintaining a "current" pointer to traverse a linked structure.
- *Divide-and-conquerer*: decompose a problem into smaller versions of itself plus some extra work to fix everything up.

When solving a problem, you should first try to decompose your problem down into these more elementary patterns. That way you can get the "easy" parts of the problem out of the way to focus on the problem-specific stuff. Likewise, as you learn about new problems, you should be taking note of these patterns that arise so that you can reproduce them for future problems as necessary.

5.4 Exploring the Problem Space

Consulting your programming toolbox is all about organizing and accessing your prior programming knowledge to help you solve the problem at hand. However, because each problem is unique, you will inevitable need to come up with new knowledge. How can you do this in a systematic way?

My suggestion is to build a *habit of exploration* that you go to when you encounter a problem. This process has four steps:

1. Understand the problem.
2. Formalize the interface.
3. Come up with examples.
4. Solve the problem (and recur).

The first step, understanding the problems, involves identifying the high-level pieces that will go into our solution. What data must we process? Broadly, what approach will we take towards solving the problem? As you get better at designing algorithms, this first step becomes quite natural and automatic. However, as you begin your journey into algorithmic design, you should be very explicit about identifying at a high-level the data and techniques you will use to solve the problem in question.

Formalizing the interface to the problem, consists of taking the high-level concepts we identified in understanding the problem and beginning to enshrine them in code. For example, at this step, we choose the types that our data take on either primitives or new class types as necessary. Broadly, all problems that we solve will take in some input and produce some output. If we are, *e.g.*, writing a function to solve the problem, then we would codify these inputs as parameters and the output as a return value. Relevant, pre-conditions and post-conditions should also be documented in comments.

Coming up with examples of how your solution should behave on different inputs serves to purposes:

1. They help us verify that we are implementing the correct solution and,
2. They help us find patterns and insights into the problem we are trying to solve.

It is very easy to skip this, but for hard problems, I can't stress the importance of using examples to explore the problem. One of the best pieces of advice I ever received on algorithm design was the following:

> You cannot tell a computer how to a solve problem until, at a bare minimum, you know how to solve the problem yourself.

Writing input-output examples is how we learn how to solve the problem ourselves. When you do this, you should be exploring the range of inputs to the problem, in particular identifying a few common cases as well as all the relevant corner cases. Finding the corner cases is important because they help us identify base cases in our programs and test our knowledge of the problem space, *e.g.*, what happens if this function is fed an empty array or string? Identifying several common cases is also important because they help us see patterns in the code which then lead to generalizations of our solution strategy to a working algorithm.

(As an important aside, I'm drawing a subtle, yet important distinction between coming up with examples versus *tests*. A test enshrines an example into an artifact you can use to automate the process of verifying your program. If possible, you should go through process of realizing your examples as tests, *e.g.*, written in JUnit. However, you should not avoid this step just because a suitable testing environment is not available for your particular language!)

After you have gotten comfortable with the problem—you know how to solve it for yourself—you can now begin trying to understand how you might write a program to do that. To do this, you need to think of solving the program in terms of the basic programming patterns that we talked about earlier. For example, you might identify that to sort an array, you follow a "deck of cards" approach where you repeatedly find the smallest card and put it at the front of the deck. To translate this high-level approach into a computer program, we must decide what programming patterns best capture this behavior:

- Because we are performing a repetitive action over an array, *iteration* is important to traverse the array.
- *Finding the minimum element* of a subarray also seems important here.
- Finally, we likely need to *swap* elements of our array to emulate putting the minimal element at the front of the deck.

Now, our task of writing a sorting function is much easier; we now need to arrange these more elementary operations in a way that achieves our sorting behavior.

Naturally, as you get better at algorithmic design, you'll coalesce or skip some of these skips entirely. But you should do your best to follow these steps as you begin your journey into data structures and algorithms, that way you increase your consistency and get the most out of all the problems we'll explore in the reminder of the course!