1

Data Structures and Algorithms

How many cities with more than 250,000 people lie within 500 miles of Dallas, Texas? How many people in my company make over $100,000 per year? Can we connect all of our telephone customers with less than 1,000 miles of cable? To answer questions like these, it is not enough to have the necessary information. We must organize that information in a way that allows us to find the answers in time to satisfy our needs.

Representing information is fundamental to computer science. The primary purpose of most computer programs is not to perform calculations, but to store and retrieve information — usually as fast as possible. For this reason, the study of data structures and the algorithms that manipulate them is at the heart of computer science. And that is what this book is about — helping you to understand how to structure information to support efficient processing.

This book has three primary goals. The first is to present the commonly used data structures. These form a programmer's basic data structure "toolkit." For many problems, some data structure in the toolkit provides a good solution.

The second goal is to introduce the idea of tradeoffs and reinforce the concept that there are costs and benefits associated with every data structure. This is done by describing, for each data structure, the amount of space and time required for typical operations.

The third goal is to teach how to measure the effectiveness of a data structure or algorithm. Only through such measurement can you determine which data structure in your toolkit is most appropriate for a new problem. The techniques presented also allow you to judge the merits of new data structures that you or others might invent.

There are often many approaches to solving a problem. How do we choose between them? At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code, and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Ideally, the resulting program is true to both of these goals. We might say that such a program is "elegant." While the algorithms and program code examples presented here attempt to be elegant in this sense, it is not the purpose of this book to explicitly treat issues related to goal (1). These are primarily concerns of the discipline of Software Engineering. Rather, this book is mostly about issues relating to goal (2).

How do we measure efficiency? Chapter 3 describes a method for evaluating the efficiency of an algorithm or computer program, called **asymptotic analysis**. Asymptotic analysis also allows you to measure the inherent difficulty of a problem. The remaining chapters use asymptotic analysis techniques to estimate the time cost for every algorithm presented. This allows you to see how each algorithm compares to other algorithms for solving the same problem in terms of its efficiency.

This first chapter sets the stage for what is to follow, by presenting some higher-order issues related to the selection and use of data structures. We first examine the process by which a designer selects a data structure appropriate to the task at hand. We then consider the role of abstraction in program design. We briefly consider the concept of a design pattern and see some examples. The chapter ends with an exploration of the relationship between problems, algorithms, and programs.

1.1 A Philosophy of Data Structures

1.1.1 The Need for Data Structures

You might think that with ever more powerful computers, program efficiency is becoming less important. After all, processor speed and memory size still continue to improve. Won't any efficiency problem we might have today be solved by tomorrow's hardware?

As we develop more powerful computers, our history so far has always been to use that additional computing power to tackle more complex problems, be it in the form of more sophisticated user interfaces, bigger problem sizes, or new problems previously deemed computationally infeasible. More complex problems demand more computation, making the need for efficient programs even greater. Worse yet, as tasks become more complex, they become less like our everyday experience. Today's computer scientists must be trained to have a thorough understanding of the principles behind efficient program design, because their ordinary life experiences often do not apply when designing computer programs.

In the most general sense, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. More commonly, people use the term "data structure" to mean an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.

Given sufficient space to store a collection of data items, it is always possible to search for specified items within the collection, print or otherwise process the data items in any desired order, or modify the value of any particular data item. Thus, it is possible to perform all necessary operations on any data structure. However, using the proper data structure can make the difference between a program running in a few seconds and one requiring many days.

A solution is said to be **efficient** if it solves the problem within the required **resource constraints**. Examples of resource constraints include the total space available to store the data — possibly divided into separate main memory and disk space constraints — and the time allowed to perform each subtask. A solution is sometimes said to be efficient if it requires fewer resources than known alternatives, regardless of whether it meets any particular requirements. The **cost** of a solution is the amount of resources that the solution consumes. Most often, cost is measured in terms of one key resource such as time, with the implied assumption that the solution meets the other resource constraints.

It should go without saying that people write programs to solve problems. However, it is crucial to keep this truism in mind when selecting a data structure to solve a particular problem. Only by first analyzing the problem to determine the performance goals that must be achieved can there be any hope of selecting the right data structure for the job. Poor program designers ignore this analysis step and apply a data structure that they are familiar with but which is inappropriate to the problem. The result is typically a slow program. Conversely, there is no sense in adopting a complex representation to "improve" a program that can meet its performance goals when implemented using a simpler design.

When selecting a data structure to solve a problem, you should follow these steps.

1. Analyze your problem to determine the basic operations that must be supported. Examples of basic operations include inserting a data item into the data structure, deleting a data item from the data structure, and finding a specified data item.
2. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

This three-step approach to selecting a data structure operationalizes a data-centered view of the design process. The first concern is for the data and the operations to be performed on them, the next concern is the representation for those data, and the final concern is the implementation of that representation.

Resource constraints on certain key operations, such as search, inserting data records, and deleting data records, normally drive the data structure selection process. Many issues relating to the relative importance of these operations are addressed by the following three questions, which you should ask yourself whenever you must choose a data structure:

- Are all data items inserted into the data structure at the beginning, or are insertions interspersed with other operations? Static applications (where the data are loaded at the beginning and never change) typically require only simpler data structures to get an efficient implementation than do dynamic applications.
- Can data items be deleted? If so, this will probably make the implementation more complicated.
- Are all data items processed in some well-defined order, or is search for specific data items allowed? "Random access" search generally requires more complex data structures.

1.1.2 Costs and Benefits

Each data structure has associated costs and benefits. In practice, it is hardly ever true that one data structure is better than another for use in all situations. If one data structure or algorithm is superior to another in all respects, the inferior one will usually have long been forgotten. For nearly every data structure and algorithm presented in this book, you will see examples of where it is the best choice. Some of the examples might surprise you.

A data structure requires a certain amount of space for each data item it stores, a certain amount of time to perform a single basic operation, and a certain amount of programming effort. Each problem has constraints on available space and time. Each solution to a problem makes use of the basic operations in some relative proportion, and the data structure selection process must account for this. Only after a careful analysis of your problem's characteristics can you determine the best data structure for the task.

Example 1.1 A bank must support many types of transactions with its customers, but we will examine a simple model where customers wish to open accounts, close accounts, and add money or withdraw money from accounts. We can consider this problem at two distinct levels: (1) the requirements for the physical infrastructure and workflow process that the bank uses in its interactions with its customers, and (2) the requirements for the database system that manages the accounts.

The typical customer opens and closes accounts far less often than he or she accesses the account. Customers are willing to wait many minutes while accounts are created or deleted but are typically not willing to wait more than a brief time for individual account transactions such as a deposit or withdrawal. These observations can be considered as informal specifications for the time constraints on the problem.

It is common practice for banks to provide two tiers of service. Human tellers or automated teller machines (ATMs) support customer access

to account balances and updates such as deposits and withdrawals. Special service representatives are typically provided (during restricted hours) to handle opening and closing accounts. Teller and ATM transactions are expected to take little time. Opening or closing an account can take much longer (perhaps up to an hour from the customer's perspective).

From a database perspective, we see that ATM transactions do not modify the database significantly. For simplicity, assume that if money is added or removed, this transaction simply changes the value stored in an account record. Adding a new account to the database is allowed to take several minutes. Deleting an account need have no time constraint, because from the customer's point of view all that matters is that all the money be returned (equivalent to a withdrawal). From the bank's point of view, the account record might be removed from the database system after business hours, or at the end of the monthly account cycle.

When considering the choice of data structure to use in the database system that manages customer accounts, we see that a data structure that has little concern for the cost of deletion, but is highly efficient for search and moderately efficient for insertion, should meet the resource constraints imposed by this problem. Records are accessible by unique account number (sometimes called an **exact-match query**). One data structure that meets these requirements is the hash table described in Chapter 9.4. Hash tables allow for extremely fast exact-match search. A record can be modified quickly when the modification does not affect its space requirements. Hash tables also support efficient insertion of new records. While deletions can also be supported efficiently, too many deletions lead to some degradation in performance for the remaining operations. However, the hash table can be reorganized periodically to restore the system to peak efficiency. Such reorganization can occur offline so as not to affect ATM transactions.

Example 1.2 A company is developing a database system containing information about cities and towns in the United States. There are many thousands of cities and towns, and the database program should allow users to find information about a particular place by name (another example of an exact-match query). Users should also be able to find all places that match a particular value or range of values for attributes such as location or population size. This is known as a **range query**.

A reasonable database system must answer queries quickly enough to satisfy the patience of a typical user. For an exact-match query, a few seconds is satisfactory. If the database is meant to support range queries that can return many cities that match the query specification, the entire opera-

tion may be allowed to take longer, perhaps on the order of a minute. To meet this requirement, it will be necessary to support operations that process range queries efficiently by processing all cities in the range as a batch, rather than as a series of operations on individual cities.

The hash table suggested in the previous example is inappropriate for implementing our city database, because it cannot perform efficient range queries. The B^+-tree of Section 10.5.1 supports large databases, insertion and deletion of data records, and range queries. However, a simple linear index as described in Section 10.1 would be more appropriate if the database is created once, and then never changed, such as an atlas distributed on a CD or accessed from a website.

1.2 Abstract Data Types and Data Structures

The previous section used the terms "data item" and "data structure" without properly defining them. This section presents terminology and motivates the design process embodied in the three-step approach to selecting a data structure. This motivation stems from the need to manage the tremendous complexity of computer programs.

A **type** is a collection of values. For example, the Boolean type consists of the values `true` and `false`. The integers also form a type. An integer is a **simple type** because its values contain no subparts. A bank account record will typically contain several pieces of information such as name, address, account number, and account balance. Such a record is an example of an **aggregate type** or **composite type**. A **data item** is a piece of information or a record whose value is drawn from a type. A data item is said to be a **member** of a type.

A **data type** is a type together with a collection of operations to manipulate the type. For example, an integer variable is a member of the integer data type. Addition is an example of an operation on the integer data type.

A distinction should be made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: the linked list and the array-based list. The list data type can therefore be implemented using a linked list or an array. Even the term "array" is ambiguous in that it can refer either to a data type or an implementation. "Array" is commonly used in computer programming to mean a contiguous block of memory locations, where each memory location stores one fixed-length data item. By this meaning, an array is a physical data structure. However, array can also mean a logical data type composed of a (typically homogeneous) collection of data items, with each data item identified by an index number. It is possible to implement arrays in many different ways. For exam-

ple, Section 12.2 describes the data structure used to implement a sparse matrix, a large two-dimensional array that stores only a relatively few non-zero values. This implementation is quite different from the physical representation of an array as contiguous memory locations.

An **abstract data type** (ADT) is the realization of a data type as a software component. The interface of the ADT is defined in terms of a type and a set of operations on that type. The behavior of each operation is determined by its inputs and outputs. An ADT does not specify *how* the data type is implemented. These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as **encapsulation**.

A **data structure** is the implementation for an ADT. In an object-oriented language such as C++, an ADT and its implementation together make up a **class**. Each operation associated with the ADT is implemented by a **member function** or **method**. The variables that define the space required by a data item are referred to as **data members**. An **object** is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

The term "data structure" often refers to data stored in a computer's main memory. The related term **file structure** often refers to the organization of data on peripheral storage, such as a disk drive or CD.

Example 1.3 The mathematical concept of an integer, along with operations that manipulate integers, form a data type. The C++ `int` variable type is a physical representation of the abstract integer. The `int` variable type, along with the operations that act on an `int` variable, form an ADT. Unfortunately, the `int` implementation is not completely true to the abstract integer, as there are limitations on the range of values an `int` variable can store. If these limitations prove unacceptable, then some other representation for the ADT "integer" must be devised, and a new implementation must be used for the associated operations.

Example 1.4 An ADT for a list of integers might specify the following operations:

- Insert a new integer at a particular position in the list.
- Return `true` if the list is empty.
- Reinitialize the list.
- Return the number of integers currently in the list.
- Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified.

One application that makes use of some ADT might use particular member functions of that ADT more than a second application, or the two applications might have different time requirements for the various operations. These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation.

Example 1.5 Two popular implementations for large disk-based database applications are hashing (Section 9.4) and the B^+-tree (Section 10.5). Both support efficient insertion and deletion of records, and both support exact-match queries. However, hashing is more efficient than the B^+-tree for exact-match queries. On the other hand, the B^+-tree can perform range queries efficiently, while hashing is hopelessly inefficient for range queries. Thus, if the database application limits searches to exact-match queries, hashing is preferred. On the other hand, if the application requires support for range queries, the B^+-tree is preferred. Despite these performance issues, both implementations solve versions of the same problem: updating and searching a large collection of records.

The concept of an ADT can help us to focus on key issues even in non-computing applications.

Example 1.6 When operating a car, the primary activities are steering, accelerating, and braking. On nearly all passenger cars, you steer by turning the steering wheel, accelerate by pushing the gas pedal, and brake by pushing the brake pedal. This design for cars can be viewed as an ADT with operations "steer," "accelerate," and "brake." Two cars might implement these operations in radically different ways, say with different types of engine, or front- versus rear-wheel drive. Yet, most drivers can operate many different cars because the ADT presents a uniform method of operation that does not require the driver to understand the specifics of any particular engine or drive design. These differences are deliberately hidden.

The concept of an ADT is one instance of an important principle that must be understood by any successful computer scientist: managing complexity through abstraction. A central theme of computer science is complexity and techniques for handling it. Humans deal with complexity by assigning a label to an assembly of objects or concepts and then manipulating the label in place of the assembly. Cognitive psychologists call such a label a **metaphor**. A particular label might be related to other pieces of information or other labels. This collection can in turn be given a label, forming a hierarchy of concepts and labels. This hierarchy of labels allows us to focus on important issues while ignoring unnecessary details.

Example 1.7 We apply the label "hard drive" to a collection of hardware that manipulates data on a particular type of storage device, and we apply the label "CPU" to the hardware that controls execution of computer instructions. These and other labels are gathered together under the label "computer." Because even the smallest home computers today have millions of components, some form of abstraction is necessary to comprehend how a computer operates.

Consider how you might go about the process of designing a complex computer program that implements and manipulates an ADT. The ADT is implemented in one part of the program by a particular data structure. While designing those parts of the program that use the ADT, you can think in terms of operations on the data type without concern for the data structure's implementation. Without this ability to simplify your thinking about a complex program, you would have no hope of understanding or implementing it.

Example 1.8 Consider the design for a relatively simple database system stored on disk. Typically, records on disk in such a program are accessed through a buffer pool (see Section 8.3) rather than directly. Variable length records might use a memory manager (see Section 12.3) to find an appropriate location within the disk file to place the record. Multiple index structures (see Chapter 10) will typically be used to access records in various ways. Thus, we have a chain of classes, each with its own responsibilities and access privileges. A database query from a user is implemented by searching an index structure. This index requests access to the record by means of a request to the buffer pool. If a record is being inserted or deleted, such a request goes through the memory manager, which in turn interacts with the buffer pool to gain access to the disk file. A program such as this is far too complex for nearly any human programmer to keep all of the details in his or her head at once. The only way to design and implement such a program is through proper use of abstraction and metaphors. In object-oriented programming, such abstraction is handled using classes.

Data types have both a **logical** and a **physical** form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form. Figure 1.1 illustrates this relationship between logical and physical forms for data types. When you implement an ADT, you are dealing with the physical form of the associated data type. When you use an ADT elsewhere in your program, you are concerned with the associated data type's logical form. Some sections of this book focus on physical implementations for a

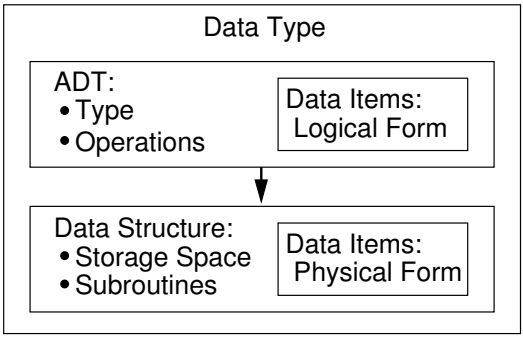

Figure 1.1 The relationship between data items, abstract data types, and data structures. The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.

given data structure. Other sections use the logical ADT for the data structure in the context of a higher-level task.

Example 1.9 A particular C++ environment might provide a library that includes a list class. The logical form of the list is defined by the public functions, their inputs, and their outputs that define the class. This might be all that you know about the list class implementation, and this should be all you need to know. Within the class, a variety of physical implementations for lists is possible. Several are described in Section 4.1.

1.3 Design Patterns

At a higher level of abstraction than ADTs are abstractions for describing the design of programs — that is, the interactions of objects and classes. Experienced software designers learn and reuse patterns for combining software components. These have come to be referred to as **design patterns**.

A design pattern embodies and generalizes important design concepts for a recurring problem. A primary goal of design patterns is to quickly transfer the knowledge gained by expert designers to newer programmers. Another goal is to allow for efficient communication between programmers. It is much easier to discuss a design issue when you share a technical vocabulary relevant to the topic.

Specific design patterns emerge from the realization that a particular design problem appears repeatedly in many contexts. They are meant to solve real problems. Design patterns are a bit like templates. They describe the structure for a design solution, with the details filled in for any given problem. Design patterns are a bit like data structures: Each one provides costs and benefits, which implies

that tradeoffs are possible. Therefore, a given design pattern might have variations on its application to match the various tradeoffs inherent in a given situation.

The rest of this section introduces a few simple design patterns that are used later in the book.

1.3.1 Flyweight

The Flyweight design pattern is meant to solve the following problem. You have an application with many objects. Some of these objects are identical in the information that they contain, and the role that they play. But they must be reached from various places, and conceptually they really are distinct objects. Because there is so much duplication of the same information, we would like to take advantage of the opportunity to reduce memory cost by sharing that space. An example comes from representing the layout for a document. The letter "C" might reasonably be represented by an object that describes that character's strokes and bounding box. However, we do not want to create a separate "C" object everywhere in the document that a "C" appears. The solution is to allocate a single copy of the shared representation for "C" objects. Then, every place in the document that needs a "C" in a given font, size, and typeface will reference this single copy. The various instances of references to a specific form of "C" are called flyweights.

We could describe the layout of text on a page by using a tree structure. The root of the tree represents the entire page. The page has multiple child nodes, one for each column. The column nodes have child nodes for each row. And the rows have child nodes for each character. These representations for characters are the flyweights. The flyweight includes the reference to the shared shape information, and might contain additional information specific to that instance. For example, each instance for "C" will contain a reference to the shared information about strokes and shapes, and it might also contain the exact location for that instance of the character on the page.

Flyweights are used in the implementation for the PR quadtree data structure for storing collections of point objects, described in Section 13.3. In a PR quadtree, we again have a tree with leaf nodes. Many of these leaf nodes represent empty areas, and so the only information that they store is the fact that they are empty. These identical nodes can be implemented using a reference to a single instance of the flyweight for better memory efficiency.

1.3.2 Visitor

Given a tree of objects to describe a page layout, we might wish to perform some activity on every node in the tree. Section 5.2 discusses tree traversal, which is the process of visiting every node in the tree in a defined order. A simple example for our text composition application might be to count the number of nodes in the tree

that represents the page. At another time, we might wish to print a listing of all the nodes for debugging purposes.

We could write a separate traversal function for each such activity that we intend to perform on the tree. A better approach would be to write a generic traversal function, and pass in the activity to be performed at each node. This organization constitutes the visitor design pattern. The visitor design pattern is used in Sections 5.2 (tree traversal) and 11.3 (graph traversal).

1.3.3 Composite

There are two fundamental approaches to dealing with the relationship between a collection of actions and a hierarchy of object types. First consider the typical procedural approach. Say we have a base class for page layout entities, with a subclass hierarchy to define specific subtypes (page, columns, rows, figures, characters, etc.). And say there are actions to be performed on a collection of such objects (such as rendering the objects to the screen). The procedural design approach is for each action to be implemented as a method that takes as a parameter a pointer to the base class type. Each action such method will traverse through the collection of objects, visiting each object in turn. Each action method contains something like a switch statement that defines the details of the action for each subclass in the collection (e.g., page, column, row, character). We can cut the code down some by using the visitor design pattern so that we only need to write the traversal once, and then write a visitor subroutine for each action that might be applied to the collection of objects. But each such visitor subroutine must still contain logic for dealing with each of the possible subclasses.

In our page composition application, there are only a few activities that we would like to perform on the page representation. We might render the objects in full detail. Or we might want a "rough draft" rendering that prints only the bounding boxes of the objects. If we come up with a new activity to apply to the collection of objects, we do not need to change any of the code that implements the existing activities. But adding new activities won't happen often for this application. In contrast, there could be many object types, and we might frequently add new object types to our implementation. Unfortunately, adding a new object type requires that we modify each activity, and the subroutines implementing the activities get rather long switch statements to distinguish the behavior of the many subclasses.

An alternative design is to have each object subclass in the hierarchy embody the action for each of the various activities that might be performed. Each subclass will have code to perform each activity (such as full rendering or bounding box rendering). Then, if we wish to apply the activity to the collection, we simply call the first object in the collection and specify the action (as a method call on that object). In the case of our page layout and its hierarchical collection of objects, those objects that contain other objects (such as a row objects that contains letters)

will call the appropriate method for each child. If we want to add a new activity with this organization, we have to change the code for every subclass. But this is relatively rare for our text compositing application. In contrast, adding a new object into the subclass hierarchy (which for this application is far more likely than adding a new rendering function) is easy. Adding a new subclass does not require changing any of the existing subclasses. It merely requires that we define the behavior of each activity that can be performed on the new subclass.

This second design approach of burying the functional activity in the subclasses is called the Composite design pattern. A detailed example for using the Composite design pattern is presented in Section 5.3.1.

1.3.4 Strategy

Our final example of a design pattern lets us encapsulate and make interchangeable a set of alternative actions that might be performed as part of some larger activity. Again continuing our text compositing example, each output device that we wish to render to will require its own function for doing the actual rendering. That is, the objects will be broken down into constituent pixels or strokes, but the actual mechanics of rendering a pixel or stroke will depend on the output device. We don't want to build this rendering functionality into the object subclasses. Instead, we want to pass to the subroutine performing the rendering action a method or class that does the appropriate rendering details for that output device. That is, we wish to hand to the object the appropriate "strategy" for accomplishing the details of the rendering task. Thus, this approach is called the Strategy design pattern.

The Strategy design pattern will be discussed further in Chapter 7. There, a sorting function is given a class (called a comparator) that understands how to extract and compare the key values for records to be sorted. In this way, the sorting function does not need to know any details of how its record type is implemented.

One of the biggest challenges to understanding design patterns is that sometimes one is only subtly different from another. For example, you might be confused about the difference between the composite pattern and the visitor pattern. The distinction is that the composite design pattern is about whether to give control of the traversal process to the nodes of the tree or to the tree itself. Both approaches can make use of the visitor design pattern to avoid rewriting the traversal function many times, by encapsulating the activity performed at each node.

But isn't the strategy design pattern doing the same thing? The difference between the visitor pattern and the strategy pattern is more subtle. Here the difference is primarily one of intent and focus. In both the strategy design pattern and the visitor design pattern, an activity is being passed in as a parameter. The strategy design pattern is focused on encapsulating an activity that is part of a larger process, so that different ways of performing that activity can be substituted. The visitor design pattern is focused on encapsulating an activity that will be performed on all

members of a collection so that completely different activities can be substituted within a generic method that accesses all of the collection members.

1.4 Problems, Algorithms, and Programs

Programmers commonly deal with problems, algorithms, and computer programs. These are three distinct concepts.

Problems: As your intuition would suggest, a **problem** is a task to be performed. It is best thought of in terms of inputs and matching outputs. A problem definition should not include any constraints on *how* the problem is to be solved. The solution method should be developed only after the problem is precisely defined and thoroughly understood. However, a problem definition should include constraints on the resources that may be consumed by any acceptable solution. For any problem to be solved by a computer, there are always such constraints, whether stated or implied. For example, any computer program may use only the main memory and disk space available, and it must run in a "reasonable" amount of time.

Problems can be viewed as functions in the mathematical sense. A **function** is a matching between inputs (the **domain**) and outputs (the **range**). An input to a function might be a single value or a collection of information. The values making up an input are called the **parameters** of the function. A specific selection of values for the parameters is called an **instance** of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

This concept of all problems behaving like mathematical functions might not match your intuition for the behavior of computer programs. You might know of programs to which you can give the same input value on two separate occasions, and two different outputs will result. For example, if you type "`date`" to a typical UNIX command line prompt, you will get the current date. Naturally the date will be different on different days, even though the same command is given. However, there is obviously more to the input for the date program than the command that you type to run the program. The date program computes a function. In other words, on any particular day there can only be a single answer returned by a properly running date program on a completely specified input. For all computer programs, the output is completely determined by the program's full set of inputs. Even a "random number generator" is completely determined by its inputs (although some random number generating systems appear to get around this by accepting a random input from a physical process beyond the user's control). The relationship between programs and functions is explored further in Section 17.3.

Algorithms: An **algorithm** is a method or a process followed to solve a problem. If the problem is viewed as a function, then an algorithm is an implementation for the function that transforms an input to the corresponding output. A problem can be solved by many different algorithms. A given algorithm solves only one problem (i.e., computes a particular function). This book covers many problems, and for several of these problems I present more than one algorithm. For the important problem of sorting I present nearly a dozen algorithms!

The advantage of knowing several solutions to a problem is that solution A might be more efficient than solution B for a specific variation of the problem, or for a specific class of inputs to the problem, while solution B might be more efficient than A for another variation or class of inputs. For example, one sorting algorithm might be the best for sorting a small collection of integers (which is important if you need to do this many times). Another might be the best for sorting a large collection of integers. A third might be the best for sorting a collection of variable-length strings.

By definition, something can only be called an algorithm if it has all of the following properties.

1. It must be *correct*. In other words, it must compute the desired function, converting each input to the correct output. Note that every algorithm implements some function, because every algorithm maps every input to some output (even if that output is a program crash). At issue here is whether a given algorithm implements the *intended* function.
2. It is composed of a series of *concrete steps*. Concrete means that the action described by that step is completely understood — and doable — by the person or machine that must perform the algorithm. Each step must also be doable in a finite amount of time. Thus, the algorithm gives us a "recipe" for solving the problem by performing a series of steps, where each such step is within our capacity to perform. The ability to perform a step can depend on who or what is intended to execute the recipe. For example, the steps of a cookie recipe in a cookbook might be considered sufficiently concrete for instructing a human cook, but not for programming an automated cookie-making factory.
3. There can be *no ambiguity* as to which step will be performed next. Often it is the next step of the algorithm description. Selection (e.g., the **`if`** statement in **C**++) is normally a part of any language for describing algorithms. Selection allows a choice for which step will be performed next, but the selection process is unambiguous at the time when the choice is made.
4. It must be composed of a *finite* number of steps. If the description for the algorithm were made up of an infinite number of steps, we could never hope to write it down, nor implement it as a computer program. Most languages for describing algorithms (including English and "pseudocode") provide some

way to perform repeated actions, known as iteration. Examples of iteration in programming languages include the **`while`** and **`for`** loop constructs of C++. Iteration allows for short descriptions, with the number of steps actually performed controlled by the input.

5. It must *terminate*. In other words, it may not go into an infinite loop.

Programs: We often think of a **computer program** as an instance, or concrete representation, of an algorithm in some programming language. In this book, nearly all of the algorithms are presented in terms of programs, or parts of programs. Naturally, there are many programs that are instances of the same algorithm, because any modern computer programming language can be used to implement the same collection of algorithms (although some programming languages can make life easier for the programmer). To simplify presentation, I often use the terms "algorithm" and "program" interchangeably, despite the fact that they are really separate concepts. By definition, an algorithm must provide sufficient detail that it can be converted into a program when needed.

The requirement that an algorithm must terminate means that not all computer programs meet the technical definition of an algorithm. Your operating system is one such program. However, you can think of the various tasks for an operating system (each with associated inputs and outputs) as individual problems, each solved by specific algorithms implemented by a part of the operating system program, and each one of which terminates once its output is produced.

To summarize: A **problem** is a function or a mapping of inputs to outputs. An **algorithm** is a recipe for solving a problem whose steps are concrete and unambiguous. Algorithms must be correct, of finite length, and must terminate for all inputs. A **program** is an instantiation of an algorithm in a programming language.

1.5 Further Reading

An early authoritative work on data structures and algorithms was the series of books *The Art of Computer Programming* by Donald E. Knuth, with Volumes 1 and 3 being most relevant to the study of data structures [Knu97, Knu98]. A modern encyclopedic approach to data structures and algorithms that should be easy to understand once you have mastered this book is *Algorithms* by Robert Sedgewick [Sed11]. For an excellent and highly readable (but more advanced) teaching introduction to algorithms, their design, and their analysis, see *Introduction to Algorithms: A Creative Approach* by Udi Manber [Man89]. For an advanced, encyclopedic approach, see *Introduction to Algorithms* by Cormen, Leiserson, and Rivest [CLRS09]. Steven S. Skiena's *The Algorithm Design Manual* [Ski10] provides pointers to many implementations for data structures and algorithms that are available on the Web.

The claim that all modern programming languages can implement the same algorithms (stated more precisely, any function that is computable by one programming language is computable by any programming language with certain standard capabilities) is a key result from computability theory. For an easy introduction to this field see James L. Hein, *Discrete Structures, Logic, and Computability* [Hei09].

Much of computer science is devoted to problem solving. Indeed, this is what attracts many people to the field. *How to Solve It* by George Pólya [Pól57] is considered to be the classic work on how to improve your problem-solving abilities. If you want to be a better student (as well as a better problem solver in general), see *Strategies for Creative Problem Solving* by Folger and LeBlanc [FL95], *Effective Problem Solving* by Marvin Levine [Lev94], and *Problem Solving & Comprehension* by Arthur Whimbey and Jack Lochhead [WL99], and *Puzzle-Based Learning* by Zbigniew and Matthew Michaelewicz [MM08].

See *The Origin of Consciousness in the Breakdown of the Bicameral Mind* by Julian Jaynes [Jay90] for a good discussion on how humans use the concept of metaphor to handle complexity. More directly related to computer science education and programming, see "Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures" by Dan Aharoni [Aha00] for a discussion on moving from programming-context thinking to higher-level (and more design-oriented) programming-free thinking.

On a more pragmatic level, most people study data structures to write better programs. If you expect your program to work correctly and efficiently, it must first be understandable to yourself and your co-workers. Kernighan and Pike's *The Practice of Programming* [KP99] discusses a number of practical issues related to programming, including good coding and documentation style. For an excellent (and entertaining!) introduction to the difficulties involved with writing large programs, read the classic *The Mythical Man-Month: Essays on Software Engineering* by Frederick P. Brooks [Bro95].

If you want to be a successful **C**++ programmer, you need good reference manuals close at hand. The standard reference for **C**++ is *The C++ Programming Language* by Bjarne Stroustrup [Str00], with further information provided in *The Annotated C++ Reference Manual* by Ellis and Stroustrup [ES90]. No **C**++ programmer should be without Stroustrup's book, as it provides the definitive description of the language and also includes a great deal of information about the principles of object-oriented design. Unfortunately, it is a poor text for learning how to program in **C**++. A good, gentle introduction to the basics of the language is Patrick Henry Winston's *On to C++* [Win94]. A good introductory teaching text for a wider range of **C**++ is Deitel and Deitel's *C++ How to Program* [DD08].

After gaining proficiency in the mechanics of program writing, the next step is to become proficient in program design. Good design is difficult to learn in any discipline, and good design for object-oriented software is one of the most difficult

of arts. The novice designer can jump-start the learning process by studying well-known and well-used design patterns. The classic reference on design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides [GHJV95] (this is commonly referred to as the "gang of four" book). Unfortunately, this is an extremely difficult book to understand, in part because the concepts are inherently difficult. A number of Web sites are available that discuss design patterns, and which provide study guides for the *Design Patterns* book. Two other books that discuss object-oriented software design are *Object-Oriented Software Design and Construction with* C++ by Dennis Kafura [Kaf98], and *Object-Oriented Design Heuristics* by Arthur J. Riel [Rie96].

1.6 Exercises

The exercises for this chapter are different from those in the rest of the book. Most of these exercises are answered in the following chapters. However, you should *not* look up the answers in other parts of the book. These exercises are intended to make you think about some of the issues to be covered later on. Answer them to the best of your ability with your current knowledge.

1.1 Think of a program you have used that is unacceptably slow. Identify the specific operations that make the program slow. Identify other basic operations that the program performs quickly enough.

1.2 Most programming languages have a built-in integer data type. Normally this representation has a fixed size, thus placing a limit on how large a value can be stored in an integer variable. Describe a representation for integers that has no size restriction (other than the limits of the computer's available main memory), and thus no practical limit on how large an integer can be stored. Briefly show how your representation can be used to implement the operations of addition, multiplication, and exponentiation.

1.3 Define an ADT for character strings. Your ADT should consist of typical functions that can be performed on strings, with each function defined in terms of its input and output. Then define two different physical representations for strings.

1.4 Define an ADT for a list of integers. First, decide what functionality your ADT should provide. Example 1.4 should give you some ideas. Then, specify your ADT in C++ in the form of an abstract class declaration, showing the functions, their parameters, and their return types.

1.5 Briefly describe how integer variables are typically represented on a computer. (Look up one's complement and two's complement arithmetic in an introductory computer science textbook if you are not familiar with these.)

Why does this representation for integers qualify as a data structure as defined in Section 1.2?

1.6 Define an ADT for a two-dimensional array of integers. Specify precisely the basic operations that can be performed on such arrays. Next, imagine an application that stores an array with 1000 rows and 1000 columns, where less than 10,000 of the array values are non-zero. Describe two different implementations for such arrays that would be more space efficient than a standard two-dimensional array implementation requiring one million positions.

1.7 Imagine that you have been assigned to implement a sorting program. The goal is to make this program general purpose, in that you don't want to define in advance what record or key types are used. Describe ways to generalize a simple sorting algorithm (such as insertion sort, or any other sort you are familiar with) to support this generalization.

1.8 Imagine that you have been assigned to implement a simple sequential search on an array. The problem is that you want the search to be as general as possible. This means that you need to support arbitrary record and key types. Describe ways to generalize the search function to support this goal. Consider the possibility that the function will be used multiple times in the same program, on differing record types. Consider the possibility that the function will need to be used on different keys (possibly with the same or different types) of the same record. For example, a student data record might be searched by zip code, by name, by salary, or by GPA.

1.9 Does every problem have an algorithm?

1.10 Does every algorithm have a **C**++ program?

1.11 Consider the design for a spelling checker program meant to run on a home computer. The spelling checker should be able to handle quickly a document of less than twenty pages. Assume that the spelling checker comes with a dictionary of about 20,000 words. What primitive operations must be implemented on the dictionary, and what is a reasonable time constraint for each operation?

1.12 Imagine that you have been hired to design a database service containing information about cities and towns in the United States, as described in Example 1.2. Suggest two possible implementations for the database.

1.13 Imagine that you are given an array of records that is sorted with respect to some key field contained in each record. Give two different algorithms for searching the array to find the record with a specified key value. Which one do you consider "better" and why?

1.14 How would you go about comparing two proposed algorithms for sorting an array of integers? In particular,

(a) What would be appropriate measures of cost to use as a basis for comparing the two sorting algorithms?

(b) What tests or analysis would you conduct to determine how the two algorithms perform under these cost measures?

1.15 A common problem for compilers and text editors is to determine if the parentheses (or other brackets) in a string are balanced and properly nested. For example, the string "((())())()" contains properly nested pairs of parentheses, but the string ")()(" does not; and the string "())" does not contain properly matching parentheses.

(a) Give an algorithm that returns **`true`** if a string contains properly nested and balanced parentheses, and **`false`** if otherwise. *Hint*: At no time while scanning a legal string from left to right will you have encountered more right parentheses than left parentheses.

(b) Give an algorithm that returns the position in the string of the first offending parenthesis if the string is not properly nested and balanced. That is, if an excess right parenthesis is found, return its position; if there are too many left parentheses, return the position of the first excess left parenthesis. Return -1 if the string is properly balanced and nested.

1.16 A graph consists of a set of objects (called vertices) and a set of edges, where each edge connects two vertices. Any given pair of vertices can be connected by only one edge. Describe at least two different ways to represent the connections defined by the vertices and edges of a graph.

1.17 Imagine that you are a shipping clerk for a large company. You have just been handed about 1000 invoices, each of which is a single sheet of paper with a large number in the upper right corner. The invoices must be sorted by this number, in order from lowest to highest. Write down as many different approaches to sorting the invoices as you can think of.

1.18 How would you sort an array of about 1000 integers from lowest value to highest value? Write down at least five approaches to sorting the array. Do not write algorithms in **C**++ or pseudocode. Just write a sentence or two for each approach to describe how it would work.

1.19 Think of an algorithm to find the maximum value in an (unsorted) array. Now, think of an algorithm to find the second largest value in the array. Which is harder to implement? Which takes more time to run (as measured by the number of comparisons performed)? Now, think of an algorithm to find the third largest value. Finally, think of an algorithm to find the middle value. Which is the most difficult of these problems to solve?

1.20 An unsorted list allows for constant-time insert by adding a new element at the end of the list. Unfortunately, searching for the element with key value X requires a sequential search through the unsorted list until X is found, which on average requires looking at half the list element. On the other hand, a

sorted array-based list of n elements can be searched in $\log n$ time with a binary search. Unfortunately, inserting a new element requires a lot of time because many elements might be shifted in the array if we want to keep it sorted. How might data be organized to support both insertion and search in $\log n$ time?