Chapter 1 Abstraction and Analysis

Objectives

- To understand how programming "in the large" differs from programming "in the small."
- To understand the motivation and use for pre- and postconditions.
- To develop design and decomposition skills.
- To understand the importance of algorithm efficiency and learn how to analyze the running time of simple algorithms.

1.1 Introduction

Believe it or not, a first course in computer programming covers all the tools strictly necessary to solve any problem that can be solved with a computer. A very famous computer scientist named Alan Turing conjectured, and it is now widely accepted, that any problem solvable with computers requires only the basic statements that all computer programming languages include: decision statements (e.g., `if`), looping statements (e.g., `for` and `while`) and the ability to store and retrieve data. Since you already know about these, you may wonder what else there is to learn. That's a good question.

1.1.1 Programming in the Large

If you think of computer programming as a process similar to constructing a building, right now you have the knowledge equivalent to how to use a few tools such as a hammer, screwdriver, saw, and drill. Those might be all the tools necessary to build a house, but that does not mean you can build yourself a habitable home, let alone one that meets modern building codes. That's not to say that you can't do some useful things. You are probably capable of building benches or birdhouses, you're just not yet ready for the challenges that come with a larger project.

In programming, just as in house construction, tackling bigger projects requires additional knowledge, techniques, and skills. This book is intended to give you a solid foundation of this additional knowledge that you can build on in future courses and throughout your career. As you work your way through this material, you will be making a transition from programming "in the small" to programming "in the large."

Software projects can vary in difficulty in many ways. Obviously, they may range from the very small (e.g., a program to convert temperatures from Celsius to Fahrenheit) to the very large (e.g., a computer operating system) to anything in-between. Projects also differ widely in how mission-critical the developed systems are. A web-based diary need not be designed to the same exacting specifications as, say, an online banking system, and neither is as critical as the software controlling a medical life-support device.

There is no single property that makes any particular project "large" or "difficult." In general, though, there are a number of characteristics that distinguish real-world programming from the simpler academic exercises that you have probably seen so far. Here are some of them:

program size So far you may have written programs that comprise up to hundreds (perhaps thousands) of lines of code. It is not uncommon for real applications to have hundreds of thousands or millions of lines. For example, the Linux operating system kernel contains around six million lines of code.

single programmer vs. programming team Most of the programs you have worked on so far have probably been your own projects. However, most software today is produced by teams of developers working together. No single programmer has complete knowledge of every facet of the system.

working from scratch vs. existing code base You have probably written most of your programs pretty much starting from scratch. In real-world projects, program-

ming happens in the context of existing applications. Existing systems may be extended, borrowed from, superseded, or used in concert with new software.

system lifetime When you are first learning to program, you may write many programs just for practice. Once your program has been graded, it may not ever be looked at again. Most real software projects have extended lifetimes. While they are in use, they continue to be refined, improved, and updated.

environment complexity A small project may be written in a single programming language using a small set of standard libraries. Larger projects tend to use many languages and a vast array of supporting development tools and software libraries.

1.1.2 The Road Ahead

The fundamental problem of programming in the large is managing the associated complexity. Humans are good at keeping track of only a few things at a time. In order to deal with a complex software system, we need ways of limiting the number of details that have to be considered at any given moment. The process of ignoring some details while concentrating on those that are relevant to the problem at hand is called *abstraction*. Effective software development is an exercise in building appropriate abstractions. Therefore, we will visit the idea of abstraction frequently throughout this book.

Another important technique in coping with complexity is to reuse solutions that have been developed before. As a programmer, you will need to learn how to use various *application programming interfaces* (APIs) for the tools/libraries you will use. An API is the collection of classes and functions that a library of code provides and an explanation of how to use them (i.e., what the parameters and return types are and what they represent). For example, you have already learned some simple APIs such as the functions provided in the Python `math` module and methods for built-in data structures such as the list and dictionary. Another common example of an API is a graphical user interface (GUI) toolkit.

Most languages provide APIs for accomplishing many common tasks. APIs will vary from language to language and from one operating system to another. This book cannot possibly begin to cover even a small fraction of the APIs that you will learn and use during your career. However, by learning a few APIs and, more importantly, by learning to develop your own APIs, you will acquire the skills that will make it easy for you to master new APIs in the future.

Just as important as being able to reuse existing code through APIs is the ability to leverage existing *knowledge* of good design principles. Over the years,

computer scientists have developed algorithms for solving common problems (e.g., searching and sorting) and ways of structuring data collections that are used as the basic building blocks in most programs. In this course, you will be learning how these algorithms and data structures work so that you can write larger, more complicated programs that are well designed and maintainable using these well-understood components. Studying these existing algorithms and data structures will also help you learn how to create your own novel algorithms and data structures for the unique problems you will face in the future.

Computer scientists have also developed techniques for analyzing and classifying the efficiency of various algorithms and data structures so that you can predict whether or not a program using them will solve problems in a reasonable amount of time and within the memory constraints you have. Naturally, you will also need to learn algorithm analysis techniques so that you can analyze the efficiency of the algorithms you invent.

This book covers abstraction and data structures using two different programming languages. Getting experience in more than one language is important for a number of reasons. Seeing how languages differ, you can start to gain an appreciation of how different tools available to the developer are suitable for different tasks. Having a larger toolkit at your disposal makes it easier to solve a wider variety of problems. However, the most important advantage is that you will also see how the underlying ideas of abstraction, reuse, and analysis are applied in both languages. Only by seeing different approaches can you really appreciate what are underlying principles versus what are just details of a particular language. Rest assured, those underlying principles will be useful no matter what languages or environments you may have in your future.

Speaking of programming languages, at about the time this book is going to press, a new version of Python is coming out (Python 3.0). The new version includes significant redesign and will not be backward compatible with programs written for the 2.x versions of Python. The code in this book has been written in Python 2.x style. As much as possible, we have tried to use conventions and features that are also compatible with Python 3.0, and the conversion to 3.0 is straightforward. To make the code run in Python 3.0, you need to keep the following changes in mind.

- `print` becomes a function call. You must put parentheses around the sequence of expressions to print.
- The `input` function acts like the old `raw_input`. If you want to evaluate user input, you must do it yourself explicitly (`eval(input("Enter a number: "))`).

- `range` no longer produces a list. You can still use it as before in `for` loops (e.g., `for i in range(10):`), but you need to use something like `nums = list(range(10))` to produce an explicit list.
- The single slash operator, `/`, always produces floating point division. Use the double slash, `//`, for integer division (this also works in Python 2.x).

We have provided both Python 2.x and Python 3.0 versions of all the code from the text in the online resources, so you should be able to use this book comfortably with any modern version of Python.

1.2 Functional Abstraction

In order to tackle a large software project, it is essential to be able to break it into smaller pieces. One way of dividing a problem into smaller pieces is to decompose it into a set of cooperating functions. This is called *functional (or procedural) abstraction.*

1.2.1 Design by Contract

To see how writing functions is an example of abstraction, let's look at a simple example. Suppose you are writing a program that needs to calculate the square root of some value. Do you know how to do this? You may or may not actually know an algorithm for computing square roots, but that really doesn't matter, because you know how to use the square root function from the Python `math` library.

```
import math
...
answer = math.sqrt(x)
```

You can use the `sqrt` function confidently, because you know *what* it does, even though you may not know exactly *how* it accomplishes that task. Thus, you are focusing on some aspects of the `sqrt` function (the what) while ignoring certain details (the how). That's abstraction.

This separation of concerns between what a component does and how it accomplishes its task is a particularly powerful form of abstraction. If we think of a function in terms of providing a service, then the programs that use the function are called *clients* of the service, and the code that actually performs the function is said to *implement* the service. A programmer working on the client needs to know

only what the function does. He or she does not need to know any of the details of how the function works. To the client, the function is like a magical black box that carries out a needed operation. Similarly, the implementer of the function does not need to worry about how the function might be used. He or she is free to concentrate only on the details of how the function accomplishes its task, ignoring the larger picture of where and why the function is actually called.

In order to accomplish this clean separation, the client and implementer must have a firm agreement about what the function is to accomplish. That is, they must have a common understanding of the *interface* between the client code and the implementation. The interface forms a sort of abstraction barrier that separates the two views of the function. Figure 1.1 illustrates the situation for the Python string split method (or the equivalent split function in the string module). The diagram shows that the function/method accepts one required parameter that is a string and one optional parameter that is a string and returns a list of strings. The client using the split function/method does not need to be concerned with how the code works (i.e., what's *inside* the box), just how to use it. What we need is a careful description of what a function will do, without having to describe how the function will accomplish the task. Such a description is called a *specification*.

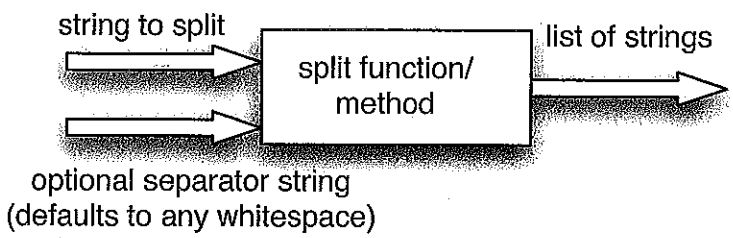

Figure 1.1: Split function as black box with interface

Obviously, one important part of a specification is describing how the function is called. That is, we need to know the name of the function, what parameters are required, and what if anything the function will return. This information is sometimes called the *signature* of a function. Beyond the signature, a specification also requires a precise description of what the function accomplishes. We need to know how the result of calling the function relates to the parameters that are provided. Sometimes, this is done rather informally. For example, suppose you are writing a `math` library function for square root. Consider this specification of the function:

```
def sqrt(x):
    """Computes the square root of x"""
```

This doesn't really do the job. The problem with such informal descriptions is that they tend to be incomplete and ambiguous. Remember, both the client and the implementer (even if they're one and the same person) should be able to fulfill their roles confidently based *only* on the specification. That's what makes the abstraction process so useful. What if the implementation *computes* the square root of x, but does not return the result? Technically, the specification is met, but the function will not be useful to the client. Is it OK if `sqrt(16)` returns `-4`? What if the implementation works only for floating-point numbers, but the client calls the function with an integer parameter? Whose fault is it then if the program crashes? What happens if the client calls this function with a negative number? Perhaps it returns a complex number as a result, or perhaps it crashes. What happens if the client calls this function with a string parameter? The bottom line is that the simple, informal description just does not tell us what to expect.

Now this may sound like nitpicking, since everyone generally "understands" what the square root function should do. If we had any questions, we could just test our assumptions by either looking at the code that implements the function or by actually trying it out (e.g., try computing `sqrt(-1)` and see what happens). But having to do either of these things breaks the abstraction barrier between the client and the implementation. Forcing the client coder to understand the actual implementation means that he or she has to wrestle with all the details of that code, thus losing the benefit of abstraction. On the other hand, if the client programmer simply relies on what the code actually does (by trying it out), he or she risks making assumptions that may not be shared by the implementer. Suppose the implementer discovers a better way of computing square roots and changes the implementation. Now the client's assumptions about certain "fringe" behavior may be incorrect. If we keep the abstraction barrier firmly in place, both the client code and the implementation can change radically; the abstraction barrier ensures that the program will continue to function properly. This desirable property is called *implementation independence.*

Hopefully you can see how precise specification of components is important when programming in the large. In most situations, careful specification is an absolute necessity; real disaster can loom when specifications are not clearly spelled out and adhered to. In one notorious example, NASA's 1999 Mars Climate Orbiter mission crashed at a loss of $125 million due to a mismatch in assumptions: a module was being given information in imperial units, but was expecting them in metric units.

Clearly, we need something better than an informal comment to have a good specification. Function specifications are often written in terms of preconditions and postconditions. A precondition of a function is a statement of what is assumed to be true about the state of the computation at the time the function is called. A postcondition is a statement about what is true after the function has finished. Here is a sample specification of the `sqrt` function using pre- and postconditions:

```
def sqrt(x):
    """Computes the square root of x.

    pre: x is an int or a float and x >= 0
    post: returns the non-negative square root of x"""
```

The job of the precondition is to state any assumptions that the implementation makes, especially those about the function parameters. In doing so, it describes the parameters using their formal names (`x` in this case). The postcondition describes whatever the code accomplishes as a function of its input parameters. Together, the pre- and postconditions describe the function as a sort of contract between the client and the implementation. If the client guarantees that the precondition is met when the function is called, then the implementation guarantees the postcondition will hold when the function terminates. For this reason, using pre- and postconditions to specify the modules of a system is sometimes called *design by contract*.

Pre- and postconditions are specific examples of a particular kind of documentation known as program *assertions*. An assertion is a statement about the state of the computation that is true at a specific point in the program. A precondition must be true just before a function executes, and the postcondition must be true immediately after. We will see later that there are other places in a program where assertions can also be extremely valuable for documentation.

If you are reading very carefully, you might be a bit uneasy about the postcondition from the `sqrt` example above. That postcondition describes what the function is supposed to do. Technically speaking, an assertion should not state what a function does, but rather what is now true at a given point in a program. It would be more correct to state the postcondition as something like `post: RETVAL` $== \sqrt{x}$, where `RETVAL` is a name used to indicate the value that was just returned by the function. Despite being less technically accurate, most programmers tend to use the less formal style of postcondition presented in our example. Given that the informal style is more popular and no less informative, we'll continue to use the "returns this, that, and the other thing" form of postcondition. Those who are sticklers for honest-to-goodness assertions can, no doubt, do the necessary translation.

This brings up an important point about pre- and postconditions in particular, and specifications in general. The whole point of a specification is that it provides a succinct and precise description of a function or other component. If the specification is ambiguous or longer or more complicated than the actual implementation code, then little has been gained. Mathematical notations tend to be succinct and exact, so they are often useful in specifications. In fact, some software engineering methods employ fully formal mathematical notations for specifying all system components. The use of these so-called *formal methods* adds precision to the development process by allowing properties of programs to be stated and proved mathematically. In the best case, one might actually be able to prove the correctness of a program, that is, that the code of a program faithfully implements its specification. Using such methods requires substantial mathematical prowess and has not been widely adopted in industry. For now, we'll stick with somewhat less formal specifications but use well-known mathematical and programming notations where they seem appropriate and helpful.

Another important consideration is where to place specifications in code. In Python, a developer has two options for placing comments into code: regular comments (indicated with a leading #) and docstrings (string expressions at the top of a module or immediately following a function or class heading). Docstrings are carried along with the objects to which they are attached and are inspectable at run-time. Docstrings are also used by the internal Python help system and by the PyDoc documentation utility. This makes docstrings a particularly good medium for specifications, since API documentation can then be created automatically using PyDoc. As a rule of thumb, docstrings should contain information that is of use to client programmers, while internal comments should be used for information that is intended only for the implementers.

1.2.2 Testing Preconditions

The basic idea of design by contract requires that if a function's precondition is met when it is called, then the postcondition must be true at the end of the function. If the precondition is not met, then all bets are off. This raises an interesting question. What should the function do when the precondition is not met? From the standpoint of the specification, it does not matter what the function does in this case, it is "off the hook," so to speak. If you are the implementer, you might be tempted to simply ignore any precondition violations. Sometimes, this means executing the function body will cause the program to immediately crash; other times the code might run, but produce nonsensical results. Neither of these outcomes seems particularly good.

A better approach is to adopt defensive programming practices. An unmet precondition indicates a mistake in the program. Rather than silently ignoring such a situation, you can detect the mistake and deal with it. But how exactly should the function do this? One idea might be to have it print an error message. The sqrt function might have some code like this:

```
def sqrt(x):
    ...
    if x < 0:
       print 'Error: can't take the square root of a negative'
    else:
       ...
```

The problem with printing an error message like this is that the calling program has no way of knowing that something has gone wrong. The output might appear, for example, in the middle of a generated report. Furthermore, the actual error message might go unnoticed. In fact, if this is a general-purpose library, it's very possible that the sqrt function is called within a GUI program, and the error message will not even appear anywhere at all.

Most of the time, it is simply not appropriate for a function that implements a service to print out messages (unless printing something is part of the specification of the method). It would be much better if the function could somehow signal that an error has occurred and then let the client program decide what to do about the problem. For some programs, the appropriate response might be to terminate the program and print an error message; in other cases, the program might be able to recover from the error. The point is that such a decision can be made only by the client.

The function could signal an error in a number of ways. Sometimes, returning an out-of-range result is used as a signal. Here's an example:

```
def sqrt(x):
    ...
    if x < 0:
       return -1
    ...
```

Since the specification of sqrt clearly implies that the return value cannot be negative, the value -1 can be used to indicate an error. Client code can check the result to see if it is OK. Another technique that is sometimes used is to have a *global* (accessible to all parts of the program) variable that records errors. The client code checks the value of this variable after each operation to see if there was an error.

Of course, the problem with this ad hoc approach to error detection is that a client program can become riddled with decision structures that constantly check to see whether an error has occurred. The logic of the code starts looking something like this:

```
x = someOperation()
if x is not OK:
    fix x
y = anotherOperation(x)
if y is not OK:
    abort
z = yetAnotherOperation(y)
if z is not OK:
    z = SOME_DEFAULT_VALUE
```

The continual error checking with each operation obscures the intent of the original algorithm.

Most modern programming languages now include *exception handling* mechanisms that provide an elegant alternative for propagating error information in a program. The basic idea behind exception handling is that program errors don't directly lead to a "crash," but rather they cause the program to transfer control to a special section called an *exception handler*. What makes this particularly useful is that the client does not have to explicitly check whether an error has occurred. The client just needs to say, in effect, "here's the code I want to execute should any errors come up." The run-time system of the language then makes sure that, should an error occur, the appropriate exception handler is called.

In Python, run-time errors generate *exception objects*. A program can include a `try` statement to catch and deal with these errors. For example, taking the square root of a negative number causes Python to generate a `ValueError`, which is a subclass of Python's general `Exception` class. If this exception is not handled by the client, it results in program termination. Here is what happens interactively:

```
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: math domain error
>>>
```

Alternatively, the program could "catch" the exception with a `try` statement:

```
>>> try:
...     sqrt(-1)
... except ValueError:
...     print "Ooops, sorry."
...
Ooops, sorry.
>>>
```

The statement(s) indented under `try` are executed, and if an error occurs, Python sees whether the error matches the type listed in any `except` clauses. The first matching `except` block is executed. If no `except` matches, then the program halts with an error message.

To take advantage of exception handling for testing preconditions, we just need to test the precondition in a decision and then generate an appropriate exception object. This is called *raising an exception* and is accomplished by the Python `raise` statement. The `raise` statement is very simple: `raise <expr>` where `<expr>` is an expression that produces an exception object containing information about what went wrong. When the `raise` statement executes, it causes the Python interpreter to interrupt the current operation and transfer control to an exception handler. If no suitable handler is found, the program will terminate.

The `sqrt` function in the Python library checks to make sure that its parameter is non-negative and also that the parameter has the correct type (either `int` or `float`). The code for `sqrt` could implement these checks as follows:

```
def sqrt(x):
    if x < 0:
        raise ValueError('math domain error')
    if type(x) not in (type(1), type(1l), type(1.0)):
        raise TypeError('number expected')

    # compute square root here
```

Notice that there are no `else`s required on these conditions. When a `raise` executes, it effectively terminates the function, so the "compute square root" portion will only execute if the preconditions are met.

Oftentimes, it is not really important what specific exception is raised when a precondition violation is detected. The important thing is that the error is diagnosed as early as possible. Python provides a statement for embedding assertions directly into code. The statement is called `assert`. It takes a Boolean expression and raises an `AssertionError` exception if the expression does not evaluate to `True`. Using `assert` makes it particularly easy to enforce preconditions.

```
def sqrt(x):
    assert x >= 0 and type(x) in (type(1), type(1l), type(1.0))
    ....
```

As you can see, the `assert` statement is a very handy way of inserting assertions directly into your code. This effectively turns the documentation of preconditions (and other assertions) into extra testing that helps to ensure that programs behave correctly, that is, according to specifications.

One potential drawback of this sort of defensive programming is that it adds extra overhead to the execution of the program. A few CPU cycles will be consumed checking the preconditions each time a function is called. However, given the ever-increasing speed of modern processors and the potential hazards of incorrect programs, that is a price that is usually well worth paying. That said, one additional benefit of the `assert` statement is that it is possible to turn off the checking of assertions, if desired. Executing Python with a -O switch on the command line causes the interpreter to skip testing of assertions. That means it is possible to have assertions on during program testing but turn them off once the system is judged to be working and placed into production.

Of course, checking assertions during testing and then turning them off in the production system is akin to practising a tightrope act 10 feet above the ground with a safety net in place and then performing the actual stunt 100 feet off the ground on a windy day—without the net. As important as it is to catch errors during testing, it's even more important to catch them when the system is in use. Our advice is to use assertions liberally and leave the checking turned on.

1.2.3 Top-Down Design

One popular technique for designing programs that you probably already know about is *top-down design*. Top-down design is essentially the direct application of functional abstraction to decompose a large problem into smaller, more manageable components. As an example, suppose you are developing a program to help your instructor with grading. Your instructor wants a program that takes a set of exam scores as input and prints out a report that summarizes student performance. Specifically, the program should report the following statistics about the data:

high score This is the largest number in the data set.

low score This is the smallest number in the data set.

mean This is the "average" score in the data set. It is often denoted $\bar{x}$ and calculated using this formula:

$$\bar{x} = \frac{\sum x_i}{n}$$

That is, we sum up all of the scores (x_i denotes the ith score) and divide by the number of scores (n).

standard deviation This is a measure of how spread out the scores are. The standard deviation, s, is given by the following formula:

$$s = \sqrt{\frac{\sum (\bar{x} - x_i)^2}{n-1}}$$

In this formula $\bar{x}$ is the mean, x_i represents the ith data value, and n is the number of data values. The formula looks complicated, but it is not hard to compute. The expression $(\bar{x} - x_i)^2$ is the square of the "deviation" of an individual item from the mean. The numerator of the fraction is the sum of the deviations (squared) across all the data values.

As a starting point for this program, you might develop a simple algorithm such as this.

```
Get scores from the user
Calculate the minimum score
Calculate the maximum score
Calculate the average (mean) score
Calculate the standard deviation
```

Suppose you are working with a friend to develop this program. You could divide this algorithm up into parts and each work on various pieces of the program. Before going off and working on the pieces, however, you will need a more complete design to ensure that the pieces that each of you develops will fit together to solve the problem. Using top-down design, each line of the algorithm can be written as a separate function. The design will just consist of the specification for each of these functions.

One obvious approach is to store the exam scores in a list that can be passed as a parameter to various functions. Using this approach, here is a sample design:

```
# stats.py
def get_scores():
    """Get scores interactively from the user

    post: returns a list of numbers obtained from user"""

def min_value(nums):
    """ find the minimum

    pre: nums is a list of numbers and len(nums) > 0
    post: returns smallest number in nums"""

def max_value(nums):
    """ find the maximum

    pre: nums is a list of numbers and len(nums) > 0
    post: returns largest number in nums"""

def average(nums):
    """ calculate the mean

    pre: nums is a list of numbers and len(nums) > 0
    post: returns the mean (a float) of the values in nums"""

def std_deviation(nums):
    """calculate the standard deviation

    pre: nums is a list of numbers and len(nums) > 1
    post: returns the standard deviation (a float) of the values
          in nums"""
```

With the specification of these functions in hand, you and your friend should easily be able to divvy up the functions and complete the program in no time. Let's implement one of the functions just to see how it might look. Here's an implementation of `std_deviation`.

```
def std_deviation(nums):

    xbar = average(nums)
    sum = 0.0
    for num in nums:
        sum += (xbar - num)**2
    return math.sqrt(sum / (len(nums) - 1))
```

Notice how this code relies on the `average` function. Since we have that function specified, we can go ahead and use it here with confidence, thus avoiding duplication

of effort. We have also used the "shorthand" += operator, which you may not have seen before. This is a convenient way of accumulating a sum. Writing `x += y` produces the same result as writing `x = x + y`.

The rest of the program is left for you to complete. As you can see, top-down design and functional specification go hand in hand. As necessary functionality is identified, a specification formalizes the design decisions so that each part can be worked on in isolation. You should have no trouble finishing up this program.

1.2.4 Documenting Side Effects

In order for specifications to be effective, they must spell out the expectations of both the client and the implementation of a service. Any effect of a service that is visible to the client should be described in the postcondition. For example, suppose that the `std_deviation` function had been implemented like this:

```
def std_deviation(nums):
    # This is bad code. Don't use it.
    xbar = average(nums)
    n = len(nums)
    sum = 0.0
    while nums != []:
        num = nums.pop()
        sum += (xbar - num)**2
    return math.sqrt(sum / (n - 1))
```

This version uses the `pop()` method of Python lists. The call to `nums.pop()` returns the last number in the list and also *deletes that item from the list.* The loop continues until all the items in the list have been processed. This version of `std_deviation` returns the correct value, so it would seem to meet the contract specified by the pre- and postconditions. However, the list object `nums` passed as a parameter is mutable, and the changes to the list will be visible to the client. The user of this code is likely to be quite surprised when they find out that calling `std_deviation(examScores)` causes all the values in `examScores` to be deleted!

These sorts of interactions between function calls and other parts of a program are called *side effects.* In this case, the deletion of items in `examScores` is a side effect of calling the `std_deviation` function. Generally, it's a good idea to avoid side effects in functions, but a strict prohibition is too strong. Some functions are designed to have side effects. The pop method of the list class is a good example. It's used in the case where one wants to get a value and also, as a side effect, remove the value from the list. What is crucial is that any side effects of a function should

be indicated in its postcondition. Since the postcondition for `std_deviation` did not say anything about nums being modified, an implementation that does this is implicitly breaking the contract. The *only* visible effects of a function should be those that are described in its postcondition.

By the way, printing something or placing information in a file are also examples of side effects. When we said above that functions should generally not print anything unless that is part of their stated functionality, we were really just identifying one special case of (potentially) undocumented side effects.

1.3 Algorithm Analysis

When we start dealing with programs that contain collections of data, we often need to know more about a function than just its pre- and postconditions. Dealing with a list of 10 or even 100 exam scores is no problem, but a list of customers for an online business might contain tens or hundreds of thousands of items. A programmer working on problems in biology might have to deal with a DNA sequence containing millions or even billions of nucleotides. Applications that search and index web pages have to deal with collections of a similar magnitude. When collection sizes get large, the efficiency of an algorithm can be just as critical as its correctness. An algorithm that gives a correct answer but requires 10 years of computing time is not likely to be very useful.

Algorithm analysis allows us to characterize algorithms according to how much time and memory they require to accomplish a task. In this section, we'll take a first look at techniques of algorithm analysis in the context of searching a collection.

1.3.1 Linear Search

Searching is the process of looking for a particular value in a collection. For example, a program that maintains the membership list for a club might need to look up the information about a particular member. This involves some form of a search process. It is a good problem for us to examine because there are numerous algorithms that can be used, and they differ in their relative efficiency.

Boiling the problem down to its simplest essence, we'll consider the problem of finding a particular number in a list. The same principles we use here will apply to more complex searching problems such as searching through a customer list to find those who live in Iowa. The specification for our simple search problem looks like this:

```
def search(items, target):
    """Locate target in items

    pre: items is a list of numbers
    post: returns non-negative x where items[x] == target, if target in
          items; returns -1, otherwise"""
```

Here are a couple interactive examples that illustrate its behavior:

```
>>> search([3, 1, 4, 2, 5], 4)
2
>>> search([3, 1, 4, 2, 5], 7)
-1
```

In the first example, the function returns the index where 4 appears in the list. In the second example, the return value -1 indicates that 7 is not in the list.

Using the built-in Python list methods, the `search` function is easily implemented:

```
# search1.py
def search(items, target):
    try:
        return items.index(target)
    except ValueError:
        return -1
```

The `index` method returns the first position in the list where a target value occurs. If `target` is not in the list, `index` raises a `ValueError` exception. In that case, we catch the exception and return -1. Clearly, this function meets the specification; the interesting question for us is how efficient is this method?

One way to determine the efficiency of an algorithm is to do empirical testing. We can simply code the algorithm and run it on different data sets to see how long it takes. A simple method for timing code in Python is to use the `time` module's `time` function, which returns the number of seconds that have passed since January 1, 1970. We can just call that method before and after our code executes and print the difference between the times. If we placed our search function in a module named `search1.py`, we could test it directly like this:

```
# time_search.py
import time
from search1 import search

items = range(1000000) # create a big list

start = time.time()
search(items, 999999) # look for the last item
stop = time.time()
print stop - start

start = time.time()
search(items, 499999) # look for the middle item
stop = time.time()
print stop - start

start = time.time()
search(items, 10)     # look for an item near the front
stop = time.time()
print stop - start
```

Try this code on your computer and note the time to search for the three numbers. What does that tell you about how the `index` method works? By the way, the Python library contains a module called `timeit` that provides a more accurate and sophisticated way of timing code. If you are doing much empirical testing, it's worth checking out this module.

Let's try our hand at developing our own search algorithm using a simple "be the computer" strategy. Suppose that I give you a page full of numbers in no particular order and ask whether the number 13 is in the list. How will you solve this problem? If you are like most people, you simply scan down the list comparing each value to 13. When you see 13 in the list, you quit and tell me that you found it. If you get to the very end of the list without seeing 13, then you tell me it's not there.

This strategy is called a *linear search*. You are searching through the list of items one by one until the target value is found. This algorithm translates directly into simple code.

```
# search2.py
def search(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1
```

You can see here that we have a simple `for` loop to go through the valid indexes for the list (`range(len(items))`). We test the item at each position to see if it is the target. If the target is found, the loop terminates by immediately returning the index of its position. If this loop goes all the way through without finding the item, the function returns `-1`.

One problem with writing the function this way is that the `range` expression creates a list of indexes that is the same size as the list being searched. Since an `int` generally requires four bytes (32 bits) of storage space, the index list in our test code would require four megabytes of memory for a list of one million numbers. In addition to the memory usage, there would also be considerable time wasted creating this second large list. Python has an alternative form of the `range` function called `xrange` that could be used instead. An `xrange` is used only for iteration, it does not actually create a list. However, the use of `xrange` is discouraged in new Python code.[1]

If your version of Python is 2.3 or newer, you can use the `enumerate` function. This elegant alternative allows you to iterate through a list and, on each iteration, you are handed the next index along with the next item. Here's how the search looks using `enumerate`.

```
# search3.py
def search(items, target):
    for i,item in enumerate(items):
        if item == target:
            return i
    return -1
```

Another approach would be to avoid the whole `range/xrange/enumerate` issue by using a `while` loop instead.

```
# search4.py
def search(items, target):
    i = 0
    while i < len(items):
        if items[i] == target:
            return i
        i += 1
    return -1
```

[1] In Python 3.0, the standard range expression behaves like xrange and does not actually create a list.

Notice that all of these search functions implement the same algorithm, namely linear search. How efficient is this algorithm? To get an idea, you might try experimenting with it. Try timing the search for the three values as you did using the list `index` method. The only code you need to change is the import of the actual `search` function, since the parameters and return values are the same. Because we wrote to a specification, the client code does not need to change, even when different implementations are mixed and matched. This is implementation independence at work. Pretty cool, huh?

1.3.2 Binary Search

The linear search algorithm was not hard to develop, and it will work very nicely for modest-sized lists. For an unordered list, this algorithm is as good as any. The Python `in` and `index` operations both implement linear searching algorithms.

If we have a very large collection of data, we might want to organize it in some way so that we don't have to look at every single item to determine where, or if, a particular value appears in the list. Suppose that the list is stored in sorted order (lowest to highest). As soon as we encounter a value that is greater than the target value, we can quit the linear search without looking at the rest of the list. On average, that saves us about half of the work. But if the list is sorted, we can do even better than this.

When a list is ordered, there is a much better searching strategy, one that you probably already know. Have you ever played the number guessing game? I pick a number between 1 and 100, and you try to guess what it is. Each time you guess, I will tell you if your guess is correct, too high, or too low. What is your strategy?

If you play this game with a very young child, they might well adopt a strategy of simply guessing numbers at random. An older child might employ a systematic approach corresponding to linear search, guessing $1, 2, 3, 4$, and so on until the mystery value is found.

Of course, virtually any adult will first guess 50. If told that the number is higher, then the range of possible values is 50–100. The next logical guess is 75. Each time we guess the middle of the remaining numbers to try to narrow down the possible range. This strategy is called a *binary search* Binary means two, and at each step, we are dividing the remaining numbers into two parts.

We can employ a binary search strategy to look through a sorted list. The basic idea is that we use two variables to keep track of the endpoints of the range in the list where the item could be. Initially, the target could be anywhere in the list, so we start with variables `low` and `high` set to the first and last positions of the list, respectively.

The heart of the algorithm is a loop that looks at the item in the middle of the remaining range to compare it to x. If x is smaller than the middle item, then we move high, so that the search is narrowed to the lower half. If x is larger, then we move low, and the search is narrowed to the upper half. The loop terminates when x is found or there are no longer any more places to look (i.e., low > high). The code below implements a binary search using our same search API.

```
# bsearch.py
def search(items, target):
    low = 0
    high = len(items) - 1
    while low <= high:          # There is still a range to search
        mid = (low + high) // 2  # position of middle item
        item = items[mid]
        if target == item :     # Found it! Return the index
            return mid
        elif target < item:     # x is in lower half of range
            high = mid - 1      #     move top marker down
        else:                   # x is in upper half
            low = mid + 1       #     move bottom marker up
    return -1                   # no range left to search,
                                # x is not there
```

This algorithm is quite a bit more sophisticated than the simple linear search. You might want to trace through a couple of sample searches to convince yourself that it actually works.

1.3.3 Informal Algorithm Comparison

So far, we have developed two very different algorithms for our simple searching problem. Which one is better? Well, that depends on what exactly we mean by better. The linear search algorithm is much easier to understand and implement. On the other hand, we expect that the binary search is more efficient, because it doesn't have to look at every value in the list. Intuitively, then, we might expect the linear search to be a better choice for small lists and binary search a better choice for larger lists. How could we actually confirm such intuitions?

One approach would be to do an empirical test. We could simply code both algorithms and try them out on various-sized lists to see how long the search takes. These algorithms are both quite short, so it would not be difficult to run a few experiments. When this test was done on one of our computers (a somewhat dated laptop), linear search was faster for lists of length 10 or less, and there was not much noticeable difference in the range of length 10–1,000. After that, binary search was

a clear winner. For a list of a million elements, linear search averaged 2.5 seconds to find a random value, whereas binary search averaged only 0.0003 seconds.

The empirical analysis has confirmed our intuition, but these are results from one particular machine under specific circumstances (amount of memory, processor speed, current load, etc.). How can we be sure that the results will always be the same?

Another approach is to analyze our algorithms abstractly to see how efficient they are. Other factors being equal, we expect the algorithm with the fewest number of "steps" to be the more efficient. But how do we count the number of steps? For example, the number of times that either algorithm goes through its main loop will depend on the particular inputs. We have already guessed that the advantage of binary search increases as the size of the list increases.

Computer scientists attack these problems by analyzing the number of steps that an algorithm will take relative to the size or difficulty of the specific problem instance being solved. For searching, the difficulty is determined by the size of the collection. Obviously, it takes more steps to find a number in a collection of a million than it does in a collection of ten. The pertinent question is how many steps are needed to find a value in a list of size n. We are particularly interested in what happens as n gets very large.

Let's consider the linear search first. If we have a list of 10 items, the most work our algorithm might have to do is to look at each item in turn. The loop will iterate at most 10 times. Suppose the list is twice as big. Then we might have to look at twice as many items. If the list is three times as large, it will take three times as long, etc. In general, the amount of time required is linearly related to the size of the list n. This is what computer scientists call a *linear time* algorithm. Now you really know why it's called a linear search.

What about the binary search? Let's start by considering a concrete example. Suppose the list contains 16 items. Each time through the loop, the remaining range is cut in half. After one pass, there are eight items left to consider. The next time through there will be four, then two, and finally one. How many times will the loop execute? It depends on how many times we can halve the range before running out of data. This table might help you to sort things out:

List Size	Halvings
1	0
2	1
4	2
8	3
16	4

Can you see the pattern here? Each extra iteration of the loop allows us to search a list that is twice as large. If the binary search loops i times, it can find a single value in a list of size 2^i. Each time through the loop, it looks at one value (the middle) in the list. To see how many items are examined in a list of size n, we need to solve this relationship: $n = 2^i$ for i. In this formula, i is just an exponent with a base of 2. Using the appropriate logarithm gives us this relationship: $i = \log_2 n$. If you are not entirely comfortable with logarithms, just remember that this value is the number of times that a collection of size n can be cut in half.

OK, so what does this bit of math tell us? Binary search is an example of a *log time* algorithm. The amount of time it takes to solve a given problem grows as the log of the problem size. In the case of binary search, each additional iteration doubles the size of the problem that we can solve.

You might not appreciate just how efficient binary search really is. Let's try to put it in perspective. Suppose you have a New York City phone book with, say, 12 million names listed in alphabetical order. You walk up to a typical New Yorker on the street and make the following proposition (assuming their number is listed): "I'm going to try guessing your name. Each time I guess a name, you tell me if your name comes alphabetically before or after the name I guess." How many guesses will you need?

Our analysis above shows the answer to this question is $log_2$12,000,000. If you don't have a calculator handy, here is a quick way to estimate the result. $2^{10} = 1{,}024$ or roughly 1,000, and $1{,}000 \times 1{,}000 = 1{,}000{,}000$. That means that $2^{10} \times 2^{10} = 2^{20} \approx 1{,}000{,}000$. 2^{20} is approximately one million. So, searching a million items requires only 20 guesses. Continuing on, we need 21 guesses for two million, 22 for four million, 23 for eight million, and 24 guesses to search among sixteen million names. We can figure out the name of a total stranger in New York City using only 24 guesses! By comparison, a linear search would require (on average) 6 million guesses. Binary search is a phenomenally good algorithm!

We said earlier that Python uses a linear search algorithm to implement its built-in searching methods. If a binary search is so much better, why doesn't Python use it? The reason is that the binary search is less general; in order to work, the list must be in order. If you want to use binary search on an unordered list, the first thing you have to do is put it in order or *sort* it. This is another well-studied problem in computer science, and one that we will return to later on.

1.3.4 Formal Analysis

In the comparison between linear and binary searches we characterized both algorithms in terms of the number of abstract steps required to solve a problem of a

certain size. We determined that linear search requires a number of steps directly proportional to the size of the list, whereas binary search requires a number of steps proportional to the (base 2) log of the list size. The nice thing about this characterization is that it tells us something about these algorithms *independent of any particular implementation.* We expect binary search to do better on large problems because it is an inherently more efficient algorithm.

When doing this kind of analysis, we are not generally concerned with the exact number of instructions an algorithm requires to solve a specific problem. This is extremely difficult to determine, since it will vary depending on the actual machine language of the computer, the language we are using to implement the algorithm, and in some cases, as we saw with the searching algorithms, the specifics of the particular input. Instead, we abstract away many issues that affect the exact running time of an implementation of an algorithm; in fact, we can ignore all the details that do not affect the relative performance of an algorithm on inputs of various sizes. Always keep in mind that our goal is to determine how the algorithm will perform on large inputs. After all, computers are fast; for small problems, efficiency is unlikely to be an issue.

To summarize, in performing algorithm analysis, we can generally make the following simplifications.

- We ignore the differences caused by using different languages and different machines to implement the algorithm.
- We ignore the differences in execution speed of various operations (i.e., we do not care that a floating-point division calculation may take longer than an integer division); we assume all "basic operations" (assignment, comparison, most mathematical operations, etc.) take the same amount of time.
- We assume all constant time operations that are independent of the input size are equivalent (i.e., we do not care if it takes 10 operations, 100 operations, or even 1,000 operations as long as those operations will solve the problem no matter what the input size is).

Obviously, each of these simplifications could make a significant difference in comparing the actual running time of two algorithms, or even two implementations of the same algorithm, but the result still shows us what to expect *as a function of the input size.* Hence, the results do tell us what kind of relative performance to expect for larger problems. Computer scientists use a notation known both as *big O* or *asymptotic* notation to specify the efficiency of an algorithm based on these simplifications.

Before looking at the details of big O notation, let's look at a couple simple mathematical functions to gain some intuition. Consider the function $f(n) = 3n^2 + 100n + 50$. Suppose you are trying to estimate the value of this function as n grows very large. You would be justified in only considering the first term. Although for smaller values of n the $100n$ term dominates, when n gets large, the contributions of the second and third term are insignificant. For example at $n = 1{,}000{,}000$ using only the first term gives a result that is within 0.01 percent of the true value of the function.

To see why the first term dominates as n increases, you just have to look at the "shape" of the graphs for the first and second terms (see Figure 1.2). Even though x is larger than x^2 over the interval from 0 to 1, x^2 overtakes it for $n > 1$. Even when we multiply x by some constant, say 100, that would change the slope of the line, since the function x^2 curves upward, it will still overtake the line for $100 * x$ (at $x = 100$). No matter what constants we multiply these functions by, the shape of the two graphs dictates that for sufficiently large values, the curve for x^2 will eventually dominate.

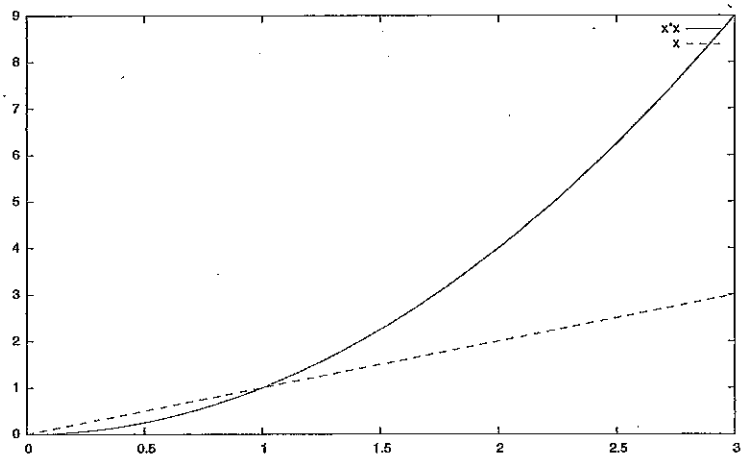

Figure 1.2: x^2 is less than x between 0 and 1, but for larger values, x^2 is greater

The idea of a dominating function is formalized in big O notation. For example, when an algorithm is said to be $O(n^2)$, it means the number of steps for the algorithm with input size n is $< cn^2$ for all $n > n_0$ for some constants c and n_0. To prove an algorithm is $O(n^2)$ we would have to find those two constants. In most cases,

it is pretty obvious (as in the examples above). What constants could we pick for the function $3n^2 + 100n + 50$? We do not need to care about having a *tight* bound. We could pick 1,000,000 for both constants since $3n^2 + 100n + 50 < 1{,}000{,}000n^2$ for every $n > 1{,}000{,}000$. If an algorithm is $2n^3$, can we find two constants to prove it is $O(n^3)$? In practice we generally do not worry about finding the constants. In most cases, it is fairly easy to convince ourselves of the relative growth rate. It should be clear that for any polynomial, it is the largest degree that matters so any polynomial of degree x is $O(n^x)$.

Now that you've seen the mathematical details, let's look at some short examples and determine the running time.

```
n = input('enter n: ')
for i in range(n):
    print i
```

This code fragment is $O(n)$. The input size, `n`, determines how many operations occur. The `print` statement will be executed `n` times. The `input` statement will be executed once. If we think about how the `for` statement works, we realize that the `range` statement generates a list of `n` items that itself takes at least `n` steps. Each time through the `for` statement, `i` is assigned to the next item in the list, so we can easily convince ourselves that there are around $2n + 1$ basic steps to execute this code. This should be enough to convince you that the algorithm is $O(n)$. We still have ignored the fact that the Python code needs to determine when the end of the list is reached, but in practice we normally do not need to go into all the details we did to convince ourselves of the running time of short code fragments.

Consider this short fragment. Can you determine its running time?

```
n = input('enter n: ')
for i in range(100):
    print i
```

With a quick look, you might be tempted to say this code is also $O(n)$ since you see a `for` loop. In this case, however, the `for` loops executes 100 times no matter what the input is. This is essentially no different than 100 `print` statements, and that is 100 constant-time operations. This code fragment runs in the same constant time regardless of the input, and we refer to all constant operations as simply $O(1)$.

Here's an example with two loops in it:

```
n = input('enter n: ')
for i in range(n):
    print i
for j in range(n):
    print j
```

These two loops execute sequentially, one right after the other. So the total running time is $O(n + n)$, which is still $O(n)$. If you find that surprising, just think of it as $O(2n)$ and remember that constant multipliers do not affect the big O notation. In general when adding sequential sections of an algorithm together, the big O for the overall algorithm is the maximum of the big Os of the individual parts. That means you just need to find the part of the algorithm that executes the most steps and analyze it.

Let's try another example with two loops.

```
n = input('enter n: ')
for i in range(n):
    for j in range(n):
        print i, j
```

In this fragment, the loops are nested. Notice that the second loop executes n times *for each iteration* of the first loop. This means the `print` statement executes a total of n^2 times, and so the code has $O(n^2)$ running time. Frequently, when you have nested loops, the running time is the product of the number of times each loop executes.

Now consider this example:

```
n = input('enter n: ')
total = 0
for i in range(n):
    for j in range(10):
        print i, j
        total = total + 1
```

Since this example also has two nested loops, you might think it is $O(n^2)$, but note that the one loop always executes 10 times no matter what the value of n is. We can still apply the rule of multiplying the number of times each loop executes; the result is $10 * n$ and that tells us this fragment is $O(n)$ (remember, constant multipliers are ignored in asymptotic analysis).

Let's try a slightly trickier case of nested loops.

```
n = input('enter n: ')
for i in range(n):
    for j in range(i, n):
        print i, j
```

Here again we have two loops nested, but the inner loop executes a different number of times during each iteration of the outer loop. Our simple multiplication rule won't work, but fortunately, the analysis is not too difficult. Remember we want to know for an input of size n, how many times does the `print` statement execute? Let's think it through. The first time through the outer loop, the inner loop executes n times. The second time it executes $n-1$ times, and so forth, until finally, on the last iteration of the outer loop, the inner loop executes 1 time. To get the total number of iterations of the inner loop, we just add these all up: $1+2+\cdots+n$.

You may have seen a formula for this sum in one of your math courses. If not, here is one way to figure it out. Suppose we add this value to itself lined up in this way:

```
      (1   +   2   +   3   + ... + n)
+     (n   + (n-1) + (n-2) + ... + 1)
```

Each column sums to $n+1$ and there are n columns. The total of all the columns is $n(n+1)$. That sum is just double the original, so dividing by 2 gives use this formula: $n(n+1)/2$. Expanding this produces a quadratic polynomial, so we can conclude this code fragment has running time $O(n^2)$.

Finally, here's a little example using a `while` loop.

```
n = input('enter n: ')
while n > 1:
    n = n // 2  # // is integer division
```

This code is a little different from all the other code fragments. We have a loop, but it does not execute n times. Each time through the loop, n is divided by 2 so we need to determine how many times it will take to reach 1. This is the same problem we examined with the "guess a number game" and binary search. The number of iterations increases by 1 each time the size of the input doubles. So the number of steps for an input of size n is represented as x in the equation $2^x = n$. The answer is $x = log_2 n$. In many algorithms, the input is divided in half and we end up with $O(log_2)$ in asymptotic notation.

Now returning to the search functions, you have all the tools you need to formally analyze the code we wrote. Our linear search uses a `for` loop that executes n times,

so it is $O(n)$. This is the reason it is referred to as a *linear search*; the running time of the function is a linear function (i.e., a polynomial of degree 1). And, as discussed earlier, the binary search algorithm for sorted lists is $O(log_2 n)$. The loop executes (at most) $log_2 n$ times and the number of operations executed each time through the loop is a constant.

Asymptotic notation tells us how efficient we can expect our algorithm to be for large data sets. For small cases or code that is only going to be executed once or twice, efficiency is often not a significant concern. Of course, if your program will take two years to solve the problem, then it is. The big O notation allows us to extrapolate and determine how long our program will take to run on a larger data set. If we want to know how long our program will take to run with an input twice as large, we can plug $2n$ in for n in our function. For example, if the analysis of an algorithm is $O(n^2)$ and we double the input, we can expect it to take four times as long, since $(2n)^2$ is $4n^2$. If it takes one minute for our algorithm to execute on an input of size one million we can expect it to take four minutes on an input of size two million.

1.3.5 Big O Notation vs. Theta Notation

Technically, big O notation gives us only an upper bound on the efficiency of an algorithm. Look back at the definition of big O. If an algorithm is $O(n)$, then it is also $O(n^2)$, $O(n^3)$, etc. In fact, we can say that most algorithms are $O(2^n)$, but that is not very useful when we want to compare two specific algorithms. Usually when we do a big O analysis of an algorithm we are trying to find a "tight" upper bound. For example, we *know* that a linear search will take twice as long to discover that a number is not in a list when the size of the list doubles. It would be more informative to say that the asymptotic growth rate of linear search is not just bounded by n, but it *is* n.

Θ is used to describe situations where we have a tight upper (and lower) bound. To formally prove an algorithm is $\Theta(f(n))$ we must find constants c_1, c_2, and n_0 such that the number of steps for the algorithm is greater than $c_1 f(n)$ and the number of steps is less than $c_2 f(n)$ for all $n > n_0$. By bounding it between two multiples of $f(n)$ we show that the number of steps grows *at the same rate as* $f(n)$ so the number of steps in the algorithm is essentially equal to some multiple of $f(n)$ (for large values of n). In practice, we will not actually find the constants unless analyzing the algorithm is particularly difficult. See Figure 1.3 for an example of bounding a function.

The growth rates of some functions that commonly appear in the analysis of algorithms are shown in Figure 1.4. Note how important the order of the algorithm

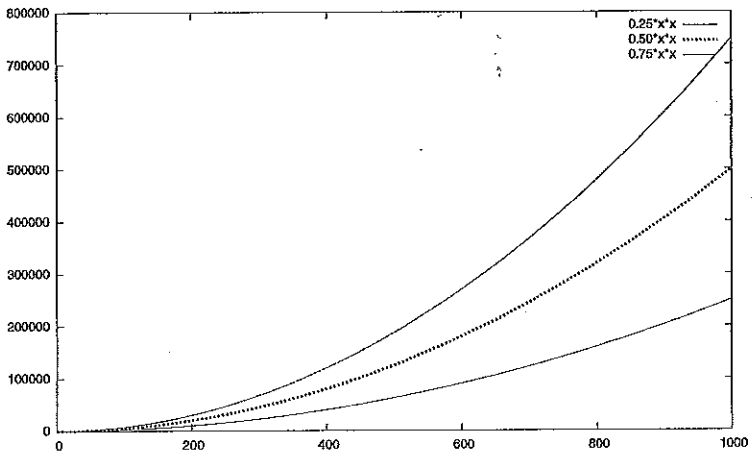

Figure 1.3: $0.5x^2$ is between $0.25x^2$ and $0.75x^2$

n	$log_2(n)$	$\sqrt{n}$	$n \log_2(n)$	n^2	n^3	2^n
100	6.6	10	660	10,000	1,000,000	10^{30}
1,000	10	32	10,000	1,000,000	10^9	10^{301}
10,000	13	100	130,000	10^8	10^{12}	10^{3010}
100,000	17	320	1,700,000	10^{10}	10^{15}	10^{30103}
1,000,000	20	1,000	$2 * 10^7$	10^{12}	10^{18}	10^{301029}

Figure 1.4: Approximate growth rate of common functions

is for making the problem solvable in a reasonable amount of time. Algorithms that have exponential growth (e.g., 2^n) cannot be used to solve problems of even modest sizes. How long would it take an exponential algorithm to complete with an input of size 100 if we can perform one billion operations per second? Using the information in Figure 1.4, we see that 2^{100} is about 10^{30} (i.e., one followed by 30 zeros); this is a very large number. Dividing it by one billion operations per second tells us it will take 10^{21} seconds or over 10^{13} years to run our algorithm on an input of size 100. The universe is thought to be between 10 and 20 billion years old so this is thousands of times longer than the universe has existed!

If we know how long it takes to solve a problem of a certain size, we can use the theta classification to approximate how long it will take to solve problems of larger sizes. For example, if we have a $\Theta(n^2)$ sorting algorithm that takes 25 seconds to

sort one million items on our computer, we can estimate how long it will take to sort two million items on our computer with the same code. This information gives us the equation $c(1{,}000{,}000)^2 = 25$ seconds. Remember that theta notation (like big O) hides the constant multiplier in front of the largest term. When setting up a specific equation, we need to include that multiplier term. We can solve for c and get $c = 2.5(10^{-11})$. We can now calculate $2.5(10^{-11})(2,000,000)^2$ and we get 100 seconds.

As you might have determined already, we do not even need to solve for c in this case. We know our algorithm is $\Theta(n^2)$ and now we want to know what happens when our input size is twice as large. We can just plug in $2n$ for n and expand: $(2n)^2 = 4n^2$. This tells us it should take four times as long to solve a problem that is twice as large when using a $\Theta(n^2)$ algorithm. This matches our earlier answer (i.e., $25(4) = 100$).

Obviously, we will try to use Θ notation whenever possible to state the performance of an algorithm. For some complicated algorithms it can be difficult to prove a tight bound and then we might just prove an upper bound (big O notation). We will also usually only analyze the worst-case running time of an algorithm. You might argue that the average case is more useful, but that is sometimes difficult to determine. For our linear search, we found the best case was $\Theta(1)$ and the worst case was $\Theta(n)$, and it is not too difficult to convince ourselves that the average case is also $\Theta(n)$. If we search once for each item in a list of unique items, the value will be found once in the first position, once in the second position, once in the third position, and so on through the last position. We know that sum is $n(n+1)/2$ and for the average case we need to divide that by the n searches we did, resulting in $\Theta(n)$. For the binary search, determining the average case is more complicated.

1.4 Chapter Summary

This chapter introduced basic concepts that are vital for writing larger software systems:

- Programming in the large varies from programming in the small along numerous dimensions. The fundamental problem in designing and implementing larger programs is how to control complexity.
- Abstraction is used to simplify and reduce the amount of information a programmer needs to understand at any given moment when writing software. One particularly useful type of abstraction (functional abstraction) allows the separation of "what" from "how" and facilitates design by contract.

- Program assertions document a program by stating what must be true at a given point of execution. Pre- and postconditions are special kinds of assertions that provide a convenient way to specify the behavior of a function or method.
- Larger problems can be broken down into smaller problems through top-down design. Specification of functional decompositions allows multiple programmers to work on a project together.
- For larger data sets, the efficiency of algorithms is important. Asymptotic analysis is used to classify the efficiency of algorithms. Big O notation is used to indicate upper bounds, while theta notation is used to characterize a more exact growth rate.

1.5 Exercises

True/False Questions

1. To use functions/classes/methods defined in a library correctly, you must understand the API (i.e., what the parameters and return values are).
2. Assuming the pre- and postconditions and code are correct, the post condition is guaranteed to be true after the code is executed if the precondition is met before the code is executed.
3. A function that detects a violation of its precondition should print out an error message.
4. A function's signature provides a complete specification of its behavior.
5. A well-designed function/method often has undocumented side effects.
6. Using the same computer, programming language, and input data, executing an algorithm that is $\Theta(n)$ must be faster than executing an algorithm that is $\Theta(n^2)$.
7. A function with more lines of code can be faster than a function with fewer lines.
8. Theta notation is an effective measure of algorithm efficiency when the expected input size for the algorithm is small.

9. All $O(n^2)$ algorithms are $\Theta(n^2)$.
10. All $\Theta(n^2)$ algorithms are $O(n^2)$.

Multiple Choice Questions

1. Which of the following is not part of the *signature* of a function?
   a) the name of the function
   b) how the function works
   c) the parameters
   d) the return value
2. Which of these actions inside a function would produce a side effect?
   a) setting an immutable parameter to a new object
   b) setting a mutable parameter to a new object
   c) modifying a mutable parameter
   d) returning a value
3. Which of the following indicates that a function's precondition was met?
   a) the function does not crash
   b) the function returns a value
   c) the function raises an exception
   d) none of the above
4. In general what will have the biggest effect on how long your algorithm takes to execute on a large data set?
   a) the efficiency of your algorithm
   b) the computer language used to implement the algorithm
   c) the number of lines of code in your algorithm
   d) the speed of the hard disk on the computer
5. A function with two loops has an asymptotic running time of
   a) $\Theta(log_2 n)$
   b) $\Theta(n)$
   c) $\Theta(n^2)$
   d) not enough information to determine
6. If a $\Theta(n^2)$ algorithm requires 3 seconds to execute on an input of one million elements, approximately how long should it take on an input of two million elements?

a) 6 seconds b) 9 seconds c) 12 seconds d) 18 seconds

7. If a $\Theta(n^3)$ algorithm requires 4 seconds to execute on an input of one million elements, approximately how long should it take on an input of two million elements?

   a) 8 seconds b) 16 seconds c) 32 seconds d) 64 seconds

8. If a $\Theta(log_2 n)$ algorithm requires 20 seconds to execute on an input of one million elements, approximately how long should it take on an input of two million elements?

   a) 21 seconds b) 25 seconds c) 30 seconds d) 40 seconds

9. If a $\Theta(2^n)$ algorithm requires 10 seconds to execute on an input of 10 elements, approximately how long should it take on an input of 20 elements?

   a) 20 seconds b) 100 seconds c) 1,000 seconds d) 10,000 seconds

10. If a computer is capable of performing one billion operations per second, approximately how long would it take to execute an algorithm that requires n^2 operations on an input of two million elements.

    a) 400 seconds
    b) 2,000 seconds
    c) 4,000 seconds
    d) 20,000 seconds

Short-Answer Questions

1. What is a side effect of a function/method?

2. Describe the basic approach of top-down design and how it relates to design by contract.

3. If you need to repeatedly search a random list of 20 items for different values that are input by the user, what search method should you use? Should you create another list that contains the same items but is sorted and search it? Why or why not?

4. If you need to repeatedly search a random list of 2,000,000 items for different values that are input by the user, what search method should you use? Should you create another list that contains the same items but is sorted and search it? Why or why not?

5. For the preceding problems is a Python list the most appropriate data type to store the numbers? If not, what Python data type would you use?

6. If a computer is capable of performing one billion operations per second, how long would it take to execute an algorithm that requires 2^n operations for an input of $n = 100$ elements?

7. If a computer is capable of performing one billion operations per second, how long would it take to execute an algorithm that requires n^2 operations on an input of $n = 1{,}000{,}000$. How long would it take if the algorithm requires n^3 operations?

8. Give a theta analysis of the time efficiency of the following code fragments.

```
a) n = input('enter n: ')
   for i in range(n):
       x = 2 * n
       while x > 1:
           x = x / 2
```

```
b) n = input('enter n: ')
   total = 0
   for i in range(n):
       for j in range(10000):
           total += j
   print total
```

```
c) total = 0
   n = input('enter n: ')
   for i in range(2 * n):
       for j in range(i, n):
           total += j
   for j in range(n):
       total += j
   print j
```

9. Our first version of the linear search algorithm used the Python `index` method and did not have any loops. Yet we said that the linear search algorithm is $\Theta(n)$. Generally, an algorithm without any loops is $\Theta(1)$. Explain the (apparent) discrepancy.

Programming Exercises

1. Create a list of one million integers numbered 0 to 999,999. Time (using the `time.time` function as we did in the examples in this chapter) the worst and best cases for the list `index` method version of the linear search, the linear search code written using a `for` statement, and the binary search code. In comments list the specifications of your computer (CPU chip and clock speed, operating system, and Python version) along with the worst and best times for each of the three searches.

2. Create a random list of 10,000, 100,000, and 1,000,000 integers with each number between 1 and 10 million. Measure how long it takes to sort each list using the built-in list's `sort` method. In comments, list the specifications of your computer (CPU chip and clock speed, operating system, and Python version) along with how long it took to sort each list. Also include comments that indicate what you think the Θ classification is for the `sort` method based on the running times.

3. The selection sort algorithm sorts a list by finding the smallest element and swapping it with the element in position zero of the list. It then finds the next smallest element and swaps it with the element in position one of the list. This process repeats until we have found the $n-1$th smallest element and put it in position $n-2$. At this point, the largest element is in position $n-1$. Implement this algorithm in Python and indicate what its Θ classification is in comments. Also time your code for the three lists described in the previous question.

4. Design your own experiment to compare the behavior of linear search and binary search on lists of various sizes. Plot your results on a graph and see if you can find a "crossover" point where linear search actually beats binary search on your computer. Since the searches will be very quick for smaller lists, you will need to be somewhat clever in how you time the searches in order to get valid data. (Hint: get larger timing intervals by timing how long it takes to do a given search many times.) Write up a complete lab report explaining your experimental set-up, methods, data, and analysis.

5. Complete the implementation of the simple statistics program in subsection 1.2.3. Be sure to thoroughly test your program on some data sets with known results.

6. Add a function to the example in subsection 1.2.3 that returns five integers: the number of scores in the 90s, in the 80s, in the 70s, in the 60s, and below

60. Be sure to provide a complete specification of your new function including appropriate pre- and postconditions along with the implementation code.

7. Whenever the average value of a set of data is needed, it is usually also appropriate to calculate the standard deviation. The current API for the simple statistics program of section subsection 1.2.3 is somewhat inefficient in this regard, as asking for both the average and the standard deviation results in the former being computed twice (why?). Redesign the API for this simple library to overcome this issue. Your new design should allow the user to efficiently calculate just the average, just the standard deviation, or both.

8. Design and implement a quiz program. The program should read question and answer information from a file. For example, a state capital quiz would contain the state and its capital on each line (e.g., `Ohio:Columbus`). Your program should ask a fixed number of questions and output the number of correct answers. Create at least three separate functions in your design.

9. Write a specification and implementation for a function that "squeezes" the duplicates out of a sorted list. For example:

```
>>> x = [1, 1, 3, 3, 3, 4, 5, 5, 8, 9, 9, 9, 9, 10]
>>> squeeze(x)
>>> x
[1, 3, 4, 5, 8, 9, 10]
```

Test your function thoroughly and analyze its theta efficiency.