Computational Complexity

2

In the last chapter we developed a drawing program. To hold the drawing commands we built the *PyList* container class which is a lot like the built-in Python list class, but helps illustrate our first data structure. When we added a drawing command to the sequence we called the append method. It turns out that this method is called a lot. In fact, the flower picture in the first chapter took around 700 commands to draw. You can imagine that a complex picture with lots of free-hand drawing could contain thousands of drawing commands. When creating a free-hand drawing we want to append the next drawing command to the sequence quickly because there are so many commands being appended. How long does it take to append a drawing command to the sequence? Can we make a guess? Should we care about the exact amount of time?

In this chapter you'll learn how to answer these questions and you'll learn what questions are important for you as a computer programmer. First you'll read about some principles of computer architecture to understand something about how long it takes a computer to do some simple operations. With that knowledge you'll have the tools you'll need to make informed decisions about how much time it might take to execute some code you have written.

2.1 Chapter Goals

By the end of this chapter you should be able to answer these questions.

- What are some of the primitive operations that a computer can perform?
- How much time does it take to perform these primitive operations?
- What does the term *computational complexity* mean?

K. D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-42209-6_2

- Why do we care about *computational complexity*?
- When do we need to be concerned about the complexity of a piece of code?
- What can we do to improve the efficiency of a piece of code?
- What is the definition of big-O notation?
- What is the definition of Theta notation?
- What is *amortized complexity* and what is its importance?
- How can we apply what we learned to make the *PyList* container class better?

2.2 Computer Architecture

A computer consists of a *Central Processing Unit* (i.e. the *CPU*) that interacts with *Input/Output* (i.e. *I/O*) devices like a keyboard, mouse, display, and network interface. When you run a program it is first read from a storage device like a hard drive into the *Random Access Memory*, or RAM, of the computer. RAM loses its contents when the power is shut off, so copies of programs are only stored in RAM while they are running. The permanent copy of a program is stored on the hard drive or some other permanent storage device.

The RAM of a computer holds a program as it is executing and also holds data that the program is manipulating. While a program is running, the CPU reads input from the input devices and stores data values in the RAM. The CPU also contains a very limited amount of memory, usually called *registers*. When an operation is performed by the CPU, such as adding two numbers together, the operands must be in registers in the CPU. Typical operations that are performed by the CPU are addition, subtraction, multiplication, division, and storing and retrieving values from the RAM.

2.2.1 Running a Program

When a user runs a program on a computer, the following actions occur:

1. The program is read from the disk or other storage device into RAM.
2. The operating system (typically Mac OS X, Microsoft Windows, or Linux) sets up two more areas of RAM called the run-time stack and the heap for use by the program.
3. The operating system starts the program executing by telling the CPU to start executing the first instruction of the computer.
4. The program reads data from the keyboard, mouse, disk, and other input sources.
5. Each instruction of the program retrieves small pieces of data from RAM, acts on them, and writes new data back to RAM.
6. Once the data is processed the result is provided as output on the screen or some other output device.

because all the people are listening, just in case their name is called. To retrieve a value, you call the name of the person and they tell you the value they were told to remember. In this way it takes exactly the same amount of time to retrieve any value from any memory location. This is how the RAM of a computer works. It takes exactly the same amount of time to store a value in any location within the RAM. Likewise, retrieving a value takes the same amount of time whether it is in the first RAM location or the last.

2.3 Accessing Elements in a Python List

With experimentation we can verify that all locations within the RAM of a computer can be accessed in the same amount of time. A Python list is a collection of contiguous memory locations. The word *contiguous* means that the memory locations of a list are grouped together consecutively in RAM. If we want to verify that the RAM of a computer behaves like a group of people all remembering their names and their values, we can run some tests with Python lists of different sizes to find the average time to retrieve from or store a value into a random element of the list.

To test the behavior of Python lists we can write a program that randomly stores and retrieves values in a list. We can test two different theories in this program.

1. The size of a list does not affect the average access time in the list.
2. The average access time at any location within a list is the same, regardless of its location within the list.

To test these two theories, we'll need to time retrieval and storing of values within a list. Thankfully, Python includes a datetime module that can be used to record the current time. By subtracting two datetime objects we can compute the number of microseconds (i.e. millionths of a second) for any operation within a program. The program in Listing 2.1 was written to test list access and record the access time for retrieving values and storing values in a Python list.

```
1  import datetime
2  import random
3  import time
4
5  def main():
6      # Write an XML file with the results
7      file = open("ListAccessTiming.xml","w")
8      file.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')
9      file.write('<Plot title="Average List Element Access Time">\n')
10
11     # Test lists of size 1000 to 200000.
12     xmin = 1000
13     xmax = 200000
14
15     # Record the list sizes in xList and the average access time within
16     # a list that size in yList for 1000 retrievals.
17     xList = []
18     yList = []
19
```

printed in the graph, the exact values are not important. What we would be interested in seeing is any trend toward longer or shorter average access times. Clearly the only trend is that the size of the list does not affect the average access time. There are some ups and downs in the experimental data, but this is caused by the system being a multi-tasking system. Another factor is likely the caching of memory locations. A cache is a way of speeding up access to memory in some situations and it is likely that the really low access times benefited from the existence of a cache for the RAM of the computer. The experimental data backs up the claim that *the size of a list does not affect the average access time in the list*.

The blue line in the plot is the result of doing 100 list retrieval and store operations on one list of 200,000 elements. The reason the blue line is higher than the red line is likely the result of doing both a retrieval from and a store operation into the element of the list. In addition, the further apart the values in memory, the less likely a cache will help reduce the access time. Whatever the reason for the blue line being higher the important thing to notice is that accessing the element at index 0 takes no more time than accessing any other element of the sequence. All locations within the list are treated equally. This backs up the claim that *the average access time at any location within a list is the same, regardless of its location within the list*.

2.4 Big-O Notation

Whichever line we look at in the experimental data, the access time never exceeds 100 μs for any of the memory accesses, even with the other things the computer might be doing. We are safe concluding that accessing memory takes less than 100 μs. In fact, 100 μs is much more time than is needed to access or store a value in a memory location. Our experimental data backs up the two claims we made earlier. However, technically, it does not prove our claim that accessing memory takes a constant amount of time. The architecture of the RAM in a computer could be examined to prove that accessing any memory location takes a constant amount of time. Accessing memory is just like calling out a name in a group of people and having that person respond with the value they were assigned. It doesn't matter which person's name is called out. The response time will be the same, or nearly the same. The actual time to access the RAM of a computer may vary a little bit if a cache is available, but at least we can say that there is an upper bound to how much time accessing a memory location will take.

This idea of an upper bound can be stated more formally. The formal statement of an upper bound is called big-O notation. The big-O refers to the Greek letter Omicron which is typically used when talking about upper bounds. As computer programmers, our number one concern is how our programs will perform when we have large amounts of data. In terms of the memory of a computer, we wanted to know how our program would perform if we have a very large list of elements. We found that all elements of a list are accessed in the same amount of time independent of how big this list is. Let's represent the size of the list by a variable called n. Let the average access time for accessing an element of a list of size n be given by $f(n)$.

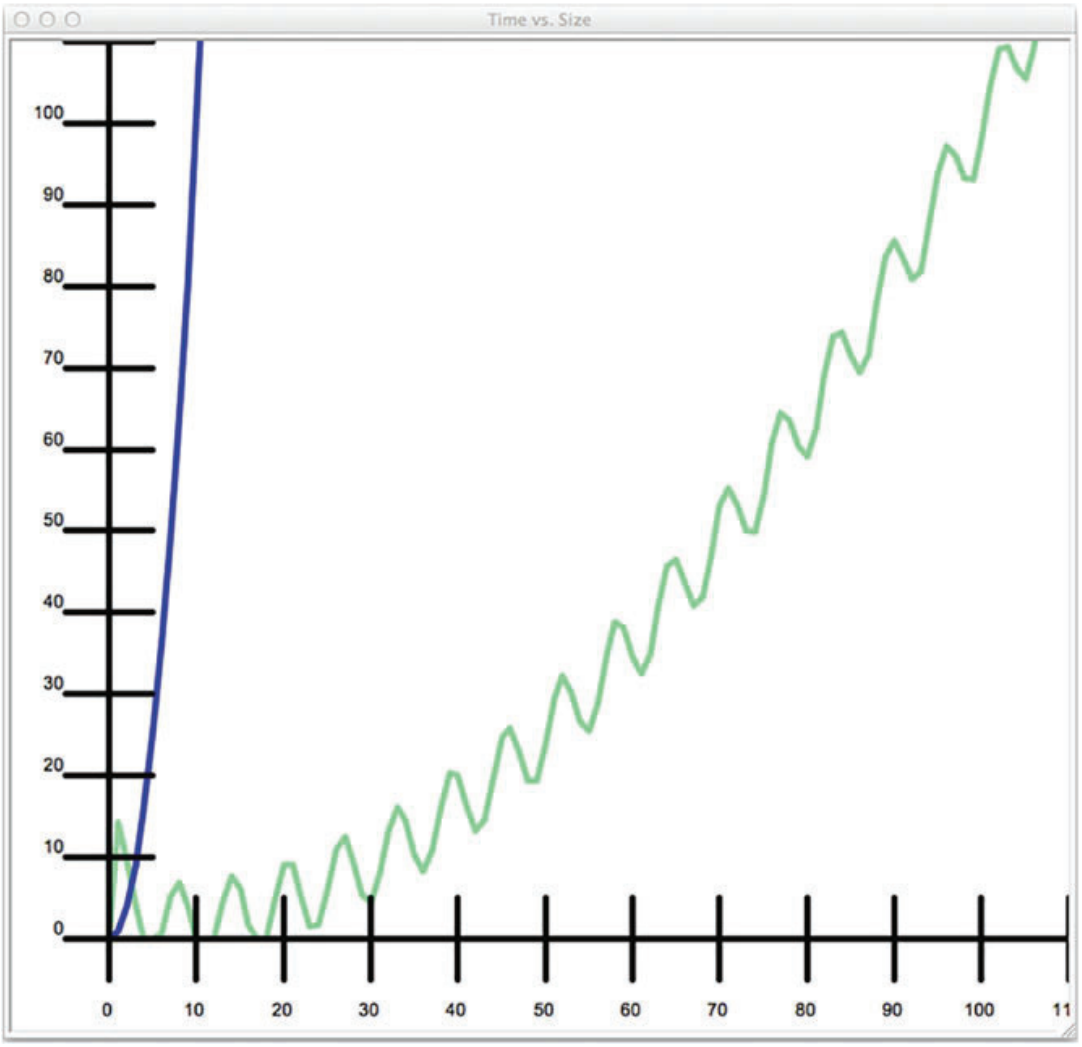

Fig. 2.3 An Upper Bound

2.5 The PyList Append Operation

We have established that accessing a memory location or storing a value in a memory location is a $O(1)$, or constant time, operation. The same goes for accessing an element of a list or storing a value in a list. The size of the list does not change the time needed to access or store an element and there is a fixed upper bound for the amount of time needed to access or store a value in memory or in a list.

With this knowledge, let's look at the drawing program again and specifically at the piece of code that appends graphics commands to the PyList. This code is used a lot in the program. Every time a new graphics command is created, it is appended to the sequence. When the user is doing some free-hand drawing, hundreds of graphics commands are getting appended every minute or so. Since free-hand drawing is somewhat compute intensive, we want this code to be as efficient as possible.

```
class PyList:
    def __init__(self):
        self.items = []

    # The append method is used to add commands to the sequence.
    def append(self,item):
        self.items = self.items + [item]

    ...
```

Listing 2.3 Inefficient Append

The code in Listing 2.3 appends a new item to the list as follows:

1. The item is made into a list by putting [and] around it. We should be careful about how we say this. The item itself is not changed. A new list is constructed from the item.
2. The two lists are concatenated together using the + operator. The + operator is an accessor method that does not change either original list. The concatenation creates a new list from the elements in the two lists.
3. The assignment of *self.items* to this new list updates the PyList object so it now refers to the new list.

The question we want to ask is, how does this append method perform as the size of the PyList grows? Let's consider the first time that the append method is called. How many elements are in the list that is referenced by self.items? Zero, right? And there is always one element in [item]. So the append method must access one element of a list to form the new list, which also has one element in it.

What happens the second time the append method is called? This time, there is one element in the list referenced by *self.items* and again one element in *[item]*. Now, two elements must be accessed to form the new list. The next time append is called three elements must be accessed to form the new list. Of course, this pattern continues for each new element that is appended to the PyList. When the nth element is appended to the sequence there will have to be n elements copied to form the new list. Overall, how many elements must be accessed to append n elements?

2.6 A Proof by Induction

We have already established that accessing each element of a list takes a constant amount of time. So, if we want to calculate the amount of time it takes to append n elements to the PyList we would have to add up all the list accesses and multiply by the amount of time it takes to access a list element plus the time it takes to store a list element. To count the total number of access and store operations we must start with the number of access and store operations for copying the list the first time an element is appended. That's one element copied. The second append requires two

Now we can use this fact in proving the equality of our original formula. Here we go!

$$\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n = \frac{n(n-1)}{2} + n = \frac{n(n-1)}{2} + \frac{2n}{2}$$

$$= \frac{n^2 - n + 2n}{2} = \frac{n^2 + n}{2} = \frac{n(n+1)}{2} \blacksquare$$

If you look at the left side and all the way over at the right side of this formula you can see the two things that we set out to prove were equal are indeed equal. This concludes our proof by induction. The meta-proof is in the formula above. It is a template that we could use to prove that the equality holds for $n = 2$. To prove the equality holds for $n = 2$ we needed to use the fact that the equality holds for $n = 1$. This was our base case. Once we have proved that it holds for $n = 2$ we could use that same formula to prove that the equality holds for $n = 3$. Mathematical induction doesn't require us to go through all the steps. As long as we've created this meta-proof we have proved that the equality holds for all n. That's the power of induction.

2.7 Making the PyList Append Efficient

Now, going back to our original problem, we wanted to find out how much time it takes to append n items to a PyList. It turns out, using the append method in Listing 2.3, it will perform in $O(n^2)$ time. This is because the first time we called append we had to copy one element of the list. The second time we needed to copy two elements. The third time append was called we needed to copy three elements. Our proof in Sect. 2.6 is that 1 + 2 + 3 + ... + n equals n*(n + 1)/2. The highest powered term in this formula is the n^2 term. Therefore, the append method in Listing 2.3 exhibits $O(n^2)$ complexity. This is not really a good result. The red curve in the graph of Fig. 2.4 shows the actual results of how much time it takes to append 200,000 elements to a *PyList*. The line looks somewhat like the graph of $f(n) = n^2$. What this tells us is that if we were to draw a complex program with say 100,000 graphics commands in it, to add one more command to the sequence it would take around 27 s. This is unacceptable! We may never draw anything that complex, but a computer should be able to add one more graphic command quicker than that!

In terms of big-O notation we say that the *append* method is $O(n^2)$. When n gets large, programs or functions with $O(n^2)$ complexity are not very good. You typically want to stay away from writing code that has this kind of computational complexity associated with it unless you are absolutely sure it will never be called on large data sizes.

One real-world example of this occurred a few years ago. A tester was testing some code and placed a CD in a CD drive. On this computer all the directories and file names on the CD were read into memory and sorted alphabetically. The sorting

The blue line in Fig. 2.4 shows how the PyList append method works when the + operator is replaced by calling the list append method instead. At 100,000 elements in the PyList we go from 27 s to add another element to maybe a second, but probably less than that. That's a nice speedup in our program. After making this change, the PyList append method is given in Listing 2.4.

```
1  class PyList:
2      def __init__(self):
3          self.items = []
4
5      # The append method is used to add commands to the sequence.
6      def append(self,item):
7          self.items.append(item)
8
9      ...
```

Listing 2.4 Efficient Append

2.8 Commonly Occurring Computational Complexities

The algorithms we will study in this text will be of one of the complexities of $O(1)$, $O(log\ n)$, $O(n\ log\ n)$, $O(n^2)$, or $O(c^n)$. A graph of the shapes of these functions appears in Fig. 2.5. Most algorithms have one of these complexities corresponding to some factor of n. Constant values added or multiplied to the terms in a formula for measuring the time needed to complete a computation do not affect the overall complexity of that operation. Computational complexity is only affected by the highest power term of the equation. The complexities graphed in Fig. 2.5 are of some power n or the log of n, except for the really awful exponential complexity of $O(c^n)$, where c is some constant value.

As you are reading the text and encounter algorithms with differing complexities, they will be one of the complexities shown in Fig. 2.5. As always, the variable n represents the size of the data provided as input to the algorithm. The time taken to process that data is the vertical axis in the graph. While we don't care about the exact numbers in this graph, we do care about the overall shape of these functions. The flatter the line, the lower the slope, the better the algorithm performs. Clearly an algorithm that has exponential complexity (i.e. $O(c^n)$) or n-squared complexity (i.e. $O(n^2)$) complexity will not perform very well except for very small values of n. If you know your algorithm will never be called for large values of n then an inefficient algorithm might be acceptable, but you would have to be really sure that you knew that your data size would always be small. Typically we want to design algorithms that are as efficient as possible.

In subsequent chapters you will encounter sorting algorithms that are $O(n^2)$ and then you'll learn that we can do better and achieve $O(n\ log\ n)$ complexity. You'll see search algorithms that are $O(n)$ and then learn how to achieve $O(log\ n)$ complexity. You'll also learn a technique called hashing that will search in $O(1)$ time. The techniques you learn will help you deal with large amounts of data as efficiently

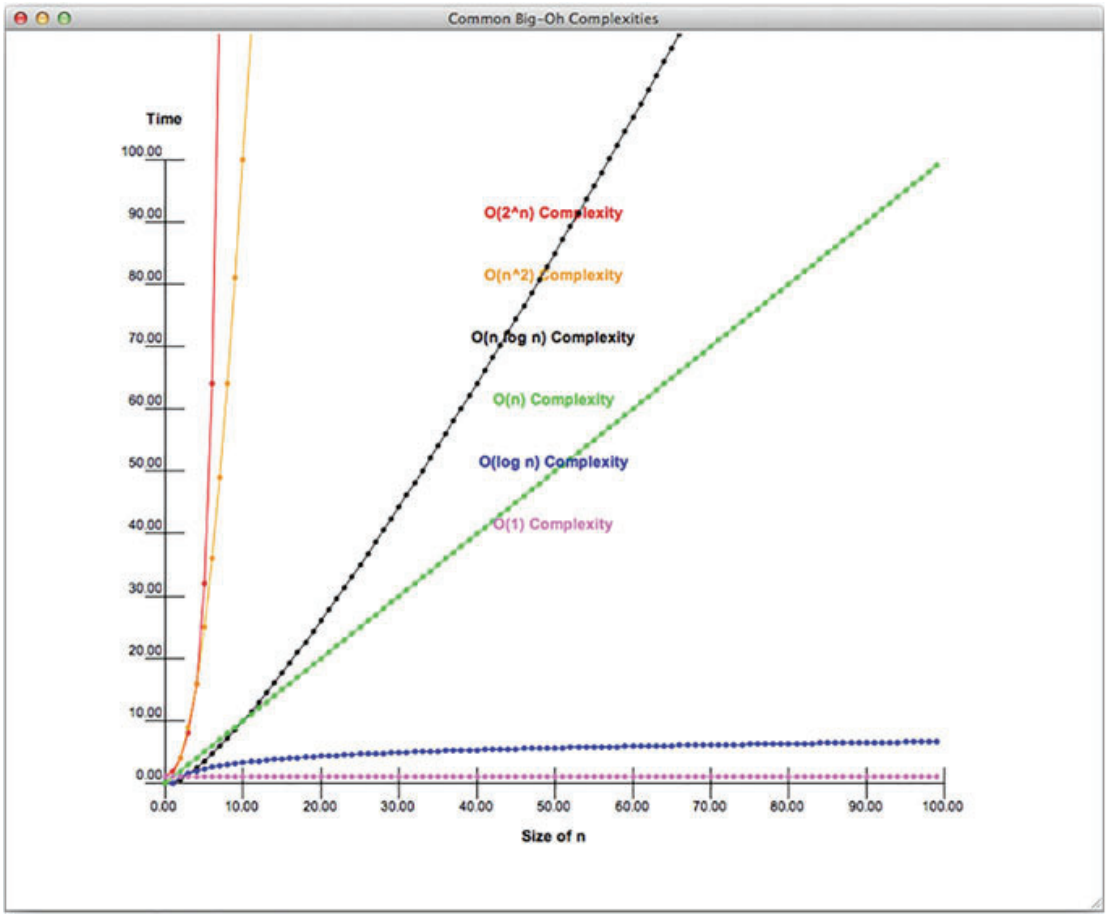

Fig. 2.5 Common Big-O Complexities

as possible. As each of these techniques are explored, you'll also have the opportunity to write some fun programs and you'll learn a good deal about object-oriented programming.

2.9 More Asymptotic Notation

Earlier in this chapter we developed big-O notation for describing an upper bound on the complexity of an algorithm. There we began with an intuitive understanding of the idea of efficiency saying that a function exhibits a complexity if it is bounded above by a function of n where n represents the size of the data given to the algorithm. In this section we further develop these concepts to bound the efficiency of an algorithm from both above and below.

We begin with an in-depth discussion of efficiency and the measurement of it in Computer Science. When concerning ourselves with algorithm efficiency there are two issues that must be considered.

- The amount of time an algorithm takes to run
- and, related to that, the amount of space an algorithm uses while running.

Typically, computer scientists will talk about a space/time tradeoff in algorithms. Sometimes we can achieve a faster running time by using more memory. But, if we use too much memory we can slow down the computer and other running programs. The *space* that is referred to is the amount of *RAM* needed to solve a problem. The *time* we are concerned with is a measure of how the number of operations grow as the size of the data grows.

Consider a function *T(n)* that is a description of the running time of an algorithm, where *n* is the size of the data given to the algorithm. As computer scientists we want to study the *asymptotic behavior* of this function. In other words, we want to study how *T(n)* increases as $n \to \infty$. The value of *n* is a Natural number representing possible sizes of input data. The natural numbers are the set of non-negative integers. The definition in Sect. 2.9.1 is a re-statement of the big-O notation definition presented earlier in this chapter.

2.9.1 Big-O Asymptotic Upper Bound

$$O(g(n)) = \{f(n) \mid \exists d > 0 \; and \; n_0 > 0 \ni 0 \leq f(n) \leq dg(n) \; \forall n \geq n_0\}$$

We write that

$$f(n) \stackrel{is}{=} O(g(n)) \Leftrightarrow f \in O((g(n)))$$

and we say that f is big-O g of n. The definition of big-O says that we can find an upper bound for the time it will take for an algorithm to run. Consider the plot of time versus data size given in Fig. 2.3. Data size, or n is the x axis, while time is the y axis.

Imagine that the green line represents the observed behavior of some algorithm. The blue line clearly is an upper bound to the green line after about $n = 4$. This is what the definition of big-O means. For a while, the upper bounding function may not be an upper bound, but eventually it becomes an upper bound and stays that way all the way to the limit as n approaches infinity.

But, does the blue line represent a tight bound on the complexity of the algorithm whose running time is depicted by the green line? We'd like to know that when we describe the complexity of an algorithm it is truly representational of the actual running time. Saying that the algorithm runs in $O(n^2)$ is accurate even if the algorithm runs in time proportional to n because big-O notation only describes an upper bound. If we truly want to say what the algorithm's running time is proportional to, then we need a little more power. This leads us to our next definition in Sect. 2.9.2.

2.9.2 Asymptotic Lower Bound

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0 \, and \, n_0 > 0 \ni 0 \leq cg(n) \leq f(n) \, \forall n \geq n_0\}$$

Omega notation serves as a way to describe a lower bound of a function. In this case the lower bound definition says for a while it might be greater, but eventually there is some n_0 where *T(n)* dominates *g(n)* for all bigger values of *n*. In that case, we can write that the algorithm is $\Omega(g(n))$. Considering our graph once again, we see that the purple line is dominated by the observed behavior sometime after $n = 75$. As with the upper bound, for a while the lower bound may be greater than the observed behavior, but after a while, the lower bound stays below the observed behavior for all bigger values of *n*.

With both a lower bound and upper bound definition, we now have the notation to define an asymptotically tight bound. This is called *Theta* notation.

2.9.3 Theta Asymptotic Tight Bound

$$\Theta(g(n)) = \{f(n) \mid \exists \, c > 0, d > 0 \, and \, n_0 > 0 \ni 0 \leq cg(n) \leq f(n) \leq dg(n) \, \forall n \geq n_0\}$$

If we can find such a function *g*, then we can declare that $\Theta(g(n))$ is an asymptotically tight bound for *T(n)*, the observed behavior of an algorithm. In Fig. 2.6 the upper bound blue line is $g(n) = n^2$ and the lower bound purple line is a plot of g(n)/110. If we let $c = 1$ and $d = 1/110$, we have the asymptotically tight bound of *T(n)* at $\Theta(n^2)$. Now, instead of saying that n-squared is an upper bound on the algorithm's behavior, we can proclaim that the algorithm truly runs in time proportional to n-squared. The behavior is bounded above and below by functions of n-squared proving the claim that the algorithm is an n-squared algorithm.

2.10 Amortized Complexity

Sometimes it is not possible to find a tight upper bound on an algorithm. For instance, most operations may be bounded by some function *c*g(n)* but every once in a while there may be an operation that takes longer. In these cases it may be helpful to employ something called *Amortized Complexity*. Amortization is a term used by accountants when spreading the cost of some business transaction over a number of years rather than applying the whole expense to the books in one fiscal year. This same idea is employed in Computer Science when the cost of an operation is averaged. The key idea behind all amortization methods is to get as tight an upper bound as we can for the worst case running time of any sequence of *n* operations on a data structure (which usually starts out empty). By dividing by *n* we get the average or *amortized* running time of each operation in the sequence.

of the list copying all items from the old list to the new list as shown in the code in Listing 2.5.

```
class PyList:
    # The size below is an initial number of locations for the list object. The
    # numItems instance variable keeps track of how many elements are currently stored
    # in the list since self.items may have empty locations at the end.
    def __init__(self,size=1):
        self.items = [None] * size
        self.numItems = 0

    def append(self,item):
        if self.numItems == len(self.items):
            # We must make the list bigger by allocating a new list and copying
            # all the elements over to the new list.
            newlst = [None] * self.numItems * 2
            for k in range(len(self.items)):
                newlst[k] = self.items[k]

            self.items = newlst

        self.items[self.numItems] = item
        self.numItems += 1

def main():
    p = PyList()

    for k in range(100):
        p.append(k)

    print(p.items)
    print(p.numItems)
    print(len(p.items))

if __name__ == "__main__":
    main()
```

Listing 2.5 A PyList Class

The claim is that, using this new PyList append method, a sequence of n append operations on a PyList object, starting with an empty list, takes $O(n)$ time meaning that individual operations must not take longer than $O(1)$ time. How can this be true? Whenever the list runs out of space a new list is allocated and all the old elements are copied to the new list. Clearly, copying n elements from one list to another takes longer than $O(1)$ time. Understanding how append could exhibit $O(1)$ complexity relies on computing the *amortized complexity* of the *append* operation. Technically, when the list size is doubled the complexity of *append* is $O(n)$. But how often does that happen? The answer is *not that often*.

2.10.1 Proof of Append Complexity

The proof that the append method has $O(1)$ complexity uses what is called the accounting method to find the amortized complexity of append. The accounting method stores up cyber dollars to pay for expensive operations later. The idea is that there must be *enough* cyber dollars to pay for any operation that is more expensive than the desired complexity.

2.11 Chapter Summary

Chapter two covered some important topics related to the efficiency of algorithms. Efficiency is an important topic because even the fastest computers will not be able to solve problems in a reasonable amount of time if the programs that are written for them are inefficient. In fact, some problems can't be solved in a reasonable amount of time no matter how the program is written. Nevertheless, it is important that we understand these issues of efficiency. Finding the complexity of a piece of code is an important skill that you will get better at the more you practice. Here are some of the things you should have learned in this chapter. You should:

- know the complexity of storing or retrieving a value from a list or the memory of the computer.
- know how memory is like a post office.
- know how memory is NOT like a post office.
- know how to use the datetime module to get information about the time it takes to complete an operation in a program.
- know how to write an XML file that can be used by the plotting program to plot information about the performance of an algorithm or piece of code.
- understand the definition of big-O notation and how it establishes an upper bound on the performance of a piece of code.
- understand why the list + operation is not as efficient as the *append* operation.
- understand the difference between $O(n)$, $O(n^2)$, and other computational complexities and why those differences are important to us as computer programmers.
- Understand Theta notation and what an asymptotically tight bound says about an algorithm.
- Understand Amortized complexity and how to apply it in some simple situations.

2.12 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. How is a list like a bunch of post office boxes?
2. How is accessing an element of a list NOT like retrieving the contents of a post office box?
3. How can you compute the amount of time it takes to complete an operation in a computer using Python?
4. In terms of computational complexity, which is better, an algorithm that is $O(n^2)$ or an algorithm that is $O(2^n)$?
5. Describe, in English, what it means for an algorithm to be $O(n^2)$.
6. When doing a proof by induction, what two parts are there to the proof?

7. If you had an algorithm with a loop that executed *n* steps the first time through, then *n–2* the second time, *n–4* the next time, and kept repeating until the last time through the loop it executed 2 steps, what would be the complexity measure of this loop? Justify your answer with what you learned in this chapter.
8. Assume you had a data set of size *n* and two algorithms that processed that data set in the same way. Algorithm A took 10 steps to process each item in the data set. Algorithm B processed each item in 100 steps. What would the complexity be of these two algorithms?
9. Explain why the *append* operation on a list is more efficient than the + operator.
10. Describe an algorithm for finding a particular value in a list. Then give the computational complexity of this algorithm. You may make any assumptions you want, but you should state your assumptions along with your algorithm.

2.13 Programming Problems

1. Devise an experiment to discover the complexity of comparing strings in Python. Does the size of the string affect the efficiency of the string comparison and if so, what is the complexity of the comparison? In this experiment you might want to consider a best case, worst case, and average case complexity. Write a program that produces an XML file with your results in the format specified in this chapter. Then use the PlotData.py program to visualize those results.
2. Conduct an experiment to prove that the product of two numbers does not depend on the size of the two numbers being multiplied. Write a program that plots the results of multiplying numbers of various sizes together. HINT: To get a good reading you may want to do more than one of these multiplications and time them as a group since a multiplication happens pretty quickly in a computer. Verify that it truly is a $O(1)$ operation. Do you see any anomalies? It might be explained by Python's support of large integers. What is the cutoff point for handling multiplications in constant time? Why? Write a program that produces an XML file with your results in the format given in this chapter. Then visualize your results with the PlotData.py program provided in this chapter.
3. Write a program to gather experimental data about comparing integers. Compare integers of different sizes and plot the amount of time it takes to do those comparisons. Plot your results by writing an XML file in the Ploy.py format. Is the comparison operation always $O(1)$? If not, can you theorize why? HINT: You may want to read about Python's support for large integers.
4. Write a short function that searches for a particular value in a list and returns the position of that value in the list (i.e. its index). Then write a program that times how long it takes to search for an item in lists of different sizes. The size of the list is your *n*. Gather results from this experiment and write them to an XML file in the PlotData.py format. What is the complexity of this algorithm? Answer this question in a comment in your program and verify that the experimental results match your prediction. Then, compare this with the *index* method on a list. Which