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?
- 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-Oh notation?

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_2

- 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 there is so little memory in the CPU, the normal mode of operation is to store values in the RAM until they are needed for a CPU operation. The RAM is a much bigger storage space than the CPU. But, because it is bigger, it is also slower than the CPU. Storing a value in RAM or retrieving a value from RAM can take as much time as several CPU operations. When needed, the values are copied from the

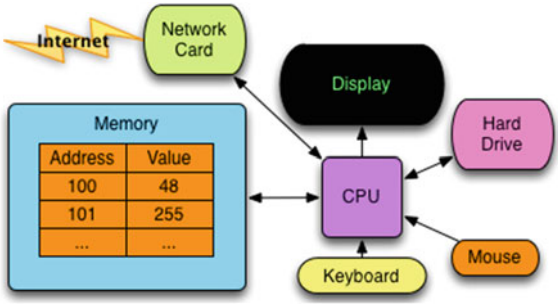

Fig. 2.1 Conceptual View of a Computer

RAM into the CPU, the operation is performed, and the result is then typically written back into the RAM. The RAM of a computer is accessed frequently as a program runs, so it is important that we understand what happens when it is accessed (Fig. 2.1).

One analogy that is often used is that of a post office. The RAM of a computer is like a collection of post office boxes. Each box has an address and can hold a value. The values you can put in RAM are called bytes (i.e. eight bits grouped together). With eight bits, 256 different values can be stored. Usually bytes are interpreted as integers, so a byte can hold values from 0 to 255. If we want to store bigger values, we can group bytes together into words. The word size of a computer is either 32 bits (i.e. four bytes) or 64 bits, depending on the architecture of the computer's hardware. All modern computer hardware is capable of retrieving or storing a word at a time.

The post office box analogy helps us to visualize how the RAM of a computer is organized, but the analogy does not serve well to show us how the RAM of a computer *behaves*. If we were going to get something from a post office box, or store something in a post office box, there would have to be some kind of search done to find the post office box first. Then the letter or letters could be placed in it or taken from it. The more post office boxes in the post office, the longer that search would take. This helps us understand the fundamental problem we study in this text. As the size of a problem space grows, how does a program or algorithm behave? In terms of this analogy, as the number of post office boxes grows, how much longer does it take to store or retrieve a value?

The RAM of a computer does not *behave* like a post office. The computer does not need to find the right RAM location before the it can retrieve or store a value. A much better analogy is a group of people, each person representing a memory location within the RAM of the computer. Each person is assigned an address or name. To store a value in a location, you call out the name of the person and then tell them what value to remember. It does not take any time to find the right person 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 Sect. 2.3.1 was written to test list access and record the access time for retrieving values and storing values in a Python list.

2.3.1 List Access Timing

```
import datetime
import random
import time

def main():

    # Write an XML file with the results
    file = open("ListAccessTiming.xml","w")

    file.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')

    file.write('<Plot title="Average List Element Access Time">\n')

    # Test lists of size 1000 to 200000.
    xmin = 1000
    xmax = 200000

    # Record the list sizes in xList and the average access time within
    # a list that size in yList for 1000 retrievals.
    xList = []
    yList = []

    for x in range(xmin, xmax+1, 1000):

        xList.append(x)

        prod = 0

```

```
        # Creates a list of size x with all 0's
        lst = [0] * x

        # let any garbage collection/memory allocation complete or at least
        # settle down
        time.sleep(1)

        # Time before the 1000 test retrievals
        starttime = datetime.datetime.now()

        for v in range(1000):
            # Find a random location within the list
            # and retrieve a value. Do a dummy operation
            # with that value to ensure it is really retrieved.
            index = random.randint(0,x-1)
            val = lst[index]
            prod = prod * val
        # Time after the 1000 test retrievals
        endtime = datetime.datetime.now()

        # The difference in time between start and end.
        deltaT = endtime - starttime

        # Divide by 1000 for the average access time
        # But also multiply by 1000000 for microseconds.
        accessTime = deltaT.total_seconds() * 1000

        yList.append(accessTime)

    file.write('  <Axes>\n')
    file.write('    <XAxis min="'+str(xmin)+'" max="'+str(xmax)+'">List Size</XAxis>\n')
    file.write('    <YAxis min="'+str(min(yList))+'" max="'+str(60)+'">Microseconds</YAxis>\n')
    file.write('  </Axes>\n')

    file.write('  <Sequence title="Average Access Time vs List Size" color="red">\n')

    for i in range(len(xList)):
        file.write('    <DataPoint x="'+str(xList[i])+'" y="'+str(yList[i])+'"/>\n')

    file.write('  </Sequence>\n')

    # This part of the program tests access at 100 random locations within a list
    # of 200,000 elements to see that all the locations can be accessed in
    # about the same amount of time.
    xList = lst
    yList = [0] * 200000

    time.sleep(2)

    for i in range(100):
        starttime = datetime.datetime.now()
        index = random.randint(0,200000-1)
        xList[index] = xList[index] + 1
        endtime = datetime.datetime.now()
        deltaT = endtime - starttime
        yList[index] = yList[index] + deltaT.total_seconds() * 1000000

    file.write('  <Sequence title="Access Time Distribution" color="blue">\n')

    for i in range(len(xList)):
        if xList[i] > 0:
            file.write('    <DataPoint x="'+str(i)+'" y="'+str(yList[i]/xList[i])+'"/>\n')

    file.write('  </Sequence>\n')
    file.write('</Plot>\n')
    file.close()

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

When running a program like this the times that you get will depend not only on the actual operations being performed, but the times will also depend on what other activity is occurring on the computer where the test is being run. All modern operating systems, like Mac OS X, Linux, or Microsoft Windows, are multi-tasking. This means the operating system can switch between tasks so that we can get email while writing a computer program, for instance. When we time something we will not only see the effects of our own program running, but all programs that are currently running on the computer. It is nearly impossible to completely isolate one program in a multi-tasking system. However, most of the time a short program will run without too much interruption.

The program in Sect. 2.3.1 writes an XML file with its results. The XML file format supports the description of experimentally collected data for a two dimensional plot of one or more sequences of data. One sample of the data that this program generates looks like Sect. 2.3.2. The data is abbreviated, but the format is as shown in Sect. 2.3.2.

2.3.2 A Plot XML Sample

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Plot title="Average List Element Access Time">
  <Axes>
    <XAxis min="1000" max="200000">List Size</XAxis>
    <YAxis min="20.244" max="60">Microseconds</YAxis>
  </Axes>
  <Sequence title="Average Access Time vs List Size" color="red">
    <DataPoint x="1000" y="33.069"/>
    <DataPoint x="2000" y="27.842"/>
    <DataPoint x="3000" y="23.908"/>
    <DataPoint x="4000" y="26.349"/>
    <DataPoint x="5000" y="23.212"/>
    <DataPoint x="6000" y="23.765"/>
    <DataPoint x="7000" y="21.251"/>
    <DataPoint x="8000" y="21.321"/>
    <DataPoint x="9000" y="23.197"/>
    <DataPoint x="10000" y="21.527"/>
    <DataPoint x="11000" y="35.799"/>
    <DataPoint x="12000" y="22.173"/>
    ...
    <DataPoint x="197000" y="26.245"/>
    <DataPoint x="198000" y="30.013"/>
    <DataPoint x="199000" y="25.888"/>
    <DataPoint x="200000" y="23.578"/>
  </Sequence>
  <Sequence title="Access Time Distribution" color="blue">
    <DataPoint x="219" y="41.0"/>
    <DataPoint x="2839" y="38.0"/>
    <DataPoint x="5902" y="38.0"/>
    <DataPoint x="8531" y="58.0"/>
    <DataPoint x="11491" y="38.0"/>
    <DataPoint x="15415" y="38.0"/>
    <DataPoint x="17645" y="31.0"/>
    <DataPoint x="18658" y="38.0"/>
    <DataPoint x="20266" y="40.0"/>
    <DataPoint x="21854" y="31.0"/>
```

```
    ...
    <DataPoint x="197159" y="37.0"/>
    <DataPoint x="199601" y="40.0"/>
  </Sequence>
</Plot>
```

Since we'll be taking a look at quite a bit of experimental data in this text, we have written a Tkinter program that will read an XML file with the format given in Sect. 2.3.2 and plot the sequences to the screen. The PlotData.py program is given in Chap. 20.4.

If we use the program to plot the data gathered by the list access experiment, we see a graph like the one in Fig. 2.2. This graph provides the experimental data to back up the two statements we made earlier about lists in Python. The red line shows the average element access time of 1,000 element accesses on a list of the given size. The average access time (computed from a sample of 1,000 random list accesses) is no longer on a list of 10,000 than it is on a list of 160,000. While the exact values are not 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

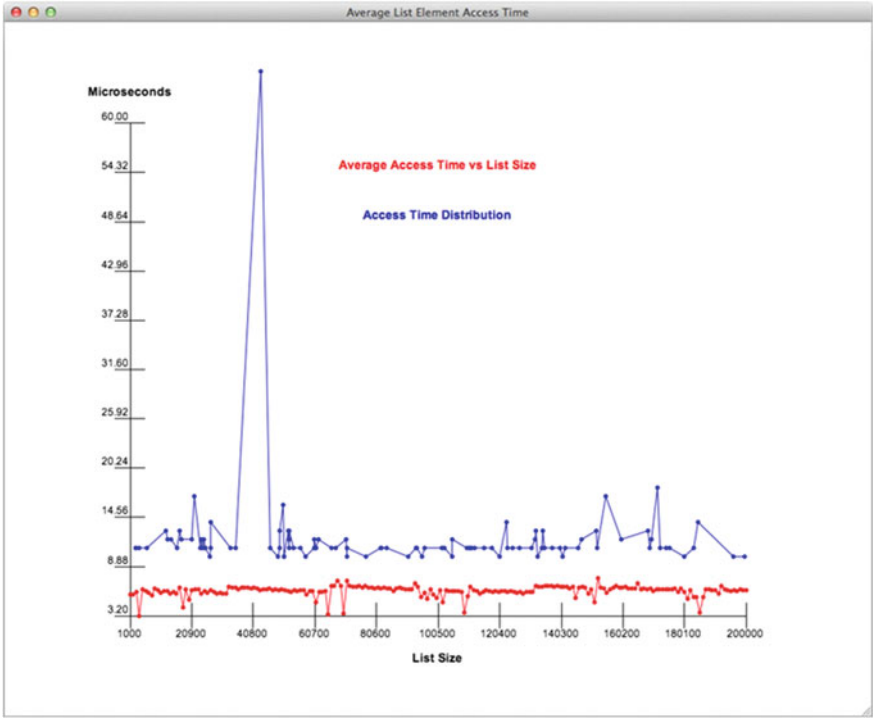

Fig. 2.2 Access Times in a Python List

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-Oh 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-Oh notation. The Big-Oh 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 $f(n)$. Now we can state the following.

$$O(g(n)) = \{f \mid \exists d > 0, n_0 \in Z^+ \ni 0 \leq f(n) \leq d\ g(n), \forall n \geq n_0\}$$

In English this reads as follows: The class of functions designated by $O(g(n))$ consists of all functions f, where there exists a d greater than 0 and an n_0 (a positive integer) such that 0 is less than or equal to $f(n)$ is less than or equal to d times $g(n)$ for all n greater than or equal to n_0.

If f is an element of O(g(n)), we say that f(n) is O(g(n)). The function g is called an asymptotic upper bound for f in this case. You may not be comfortable with the mathematical description above. Stated in English the set named O(g(n)) consists of the set of all functions, f(n), that have an upper bound of d* g(n), as n approaches infinity. This is the meaning of the word *asymptotic*. The idea of an asymptotic bound means that for some small values of n the value of f(n) might be bigger than the value of d* g(n), but once n gets big enough (i.e. bigger than n_0), then for all bigger n it will always be true that f(n) is less than d* g(n). This idea of an asymptotic upper bound is pictured in Fig. 2.3. For some smaller values the function's performance, shown in green, may be worse than the blue upper bound line, but eventually the upper bound is bigger for all larger values of n.

We have seen that the average time to access an element in a list is constant and does not depend on the list size. In the example in Fig. 2.2, the list size is the n in the definition and the average time to access an element in a list of size n is the f(n).

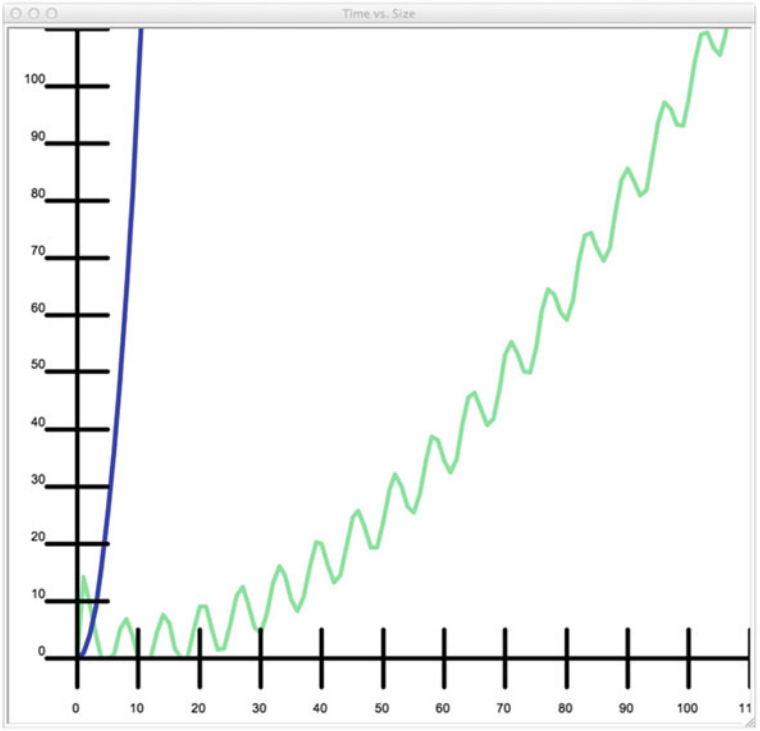

Fig. 2.3 An Upper Bound

Because the time to access an element does not depend on n, we can pick $g(\mathrm{n}) = 1$. So, we say that the average time to access an element in a list of size n is O(1). If we assume it never takes longer than 100 µs to access an element of a list in Python, then a good choice for d would be 100. According to the definition above then it must be the case that f(n) is less than or equal to 100 once n gets big enough.

The choice of $g(\mathrm{n}) = 1$ is arbitrary in computing the complexity of accessing an element of a list. We could have chosen $g(\mathrm{n}) = 2$. If $g(\mathrm{n}) = 2$ were chosen, d might be chosen to be 50 instead of 100. But, since we are only concerned with the overall growth in the function g, the choice of 1 or 2 is irrelevant and the simplest function is chosen, in this case O(1). In English, when an operation or program is O(1), we say it is a *constant time* operation or program. This means the operation does not depend on the size of n.

It turns out that most operations that a computer can perform are O(1). For instance, adding two numbers together is a O(1) operation. So is multiplication of two numbers. While both operations require several cycles in a computer, the total number of cycles does not depend on the size of the integers or floating point numbers being added or multiplied. A cycle is simply a unit of time in a computer. Comparing two values is also a constant time operation. When computing complexity, any arithmetic calculation or comparison can be considered a constant time operation.

This idea of computational complexity is especially important when the complexity of a piece of code depends on n. In the next section we'll see some code that depends on the size of the list it is working with and how important it is that we understand the implications of how we write even a small piece of code.

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.

2.5.1 Inefficient Append

```
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]

    ...
```

The code in Sect. 2.5.1 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

copy operations. The third append requires three copy operations. So, we have the following number of list elements being copied.

$$1 + 2 + 3 + 4 + \cdots + n = \sum_{i=1}^{n} i$$

In mathematics we can express this sum with a summation symbol (i.e. $\sum$). This is the mathematical way of expressing the sum of the first n integers. But, what is this equal to? It turns out with a little work, we can find that the following is true.

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

We can prove this is true using a proof technique from Mathematics called mathematical induction. There are a couple of variations of mathematical induction. We'll use what is called weak induction to prove this. When proving something using induction you are really constructing a meta-proof. A meta-proof is a set of steps that you can repeat over and over again to find your desired result. The power of induction is that once we have constructed the meta-proof, we have proved that the result is true for all possible values of n.

We want to prove that the formula given above is valid for all n. To do this we first show it is true for a simple value of n. In our case we'll pick 1 as our value of n. In that case we have the following.

$$\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$$

This is surely true. This step is called the *base case* of the inductive proof. Every *proof by induction* must have a base case and it is usually trivial.

The next step is to create the meta-proof. This meta-proof is called the *inductive case*. When forming the inductive case we get to assume that the formula holds for all values, m, where m is less than n. This is called *strong induction*. In *weak induction* we get to assume that the formula is valid for $n-1$ and we want to show that it is valid for n. We'll use weak induction in this problem to finish our proof. Again, this step helps us form a set of steps that we can apply over and over again to get from our base case to whatever value of n we need to find. To begin we will make note of the following.

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

This is true by the definition of summation. But now we have a sum that goes to $n-1$ and weak induction says that we know the equation is valid for $n-1$. This is called the *inductive hypothesis*. Since it holds for $n-1$ we know the following is true. We get this by substituting $n-1$ everyplace that we see an n in the original formula.

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

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

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

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 Sect. 2.5.1, 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 $1 + 2 + 3 + \cdots + n$ equals $n*(n + 1)/2$. The highest powered term in this formula is the n^2 term. Therefore, the append method in Sect. 2.5.1 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-Oh 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 algorithm that was used in that case had $O(n^2)$ complexity. This was OK because most CDs put in this computer had a relatively small number of directories and files on them. However, along came one CD with literally hundreds of thousands of files

on it. The computer did nothing but sort those file names alphabetically for around 12 h. When this was discovered, the programmer rewrote the sorting code to be more efficient and reduced the sorting time to around 15 s. That's a BIG difference! It also illustrates just how important this idea of computational complexity is.

If we take another look at our PyList append method we might be able to make it more efficient if we didn't have to access each element of the first list when concatenating the two lists. The use of the + operator is what causes Python to access each element of that first list. When + is used a new list is created with space for one more element. Then all the elements from the old list must be copied to the new list and the new element is added at the end of this list.

Using the *append* method on lists changes the code to use a mutator method to alter the list by adding just one more element. It turns out that adding one more element to an already existing list is very efficient in Python. In fact, appending an item to a list is a O(1) operation as we'll see later in this chapter. This means to append n items to a list we have gone from $O(n^2)$ to O(n). Later in this chapter we'll learn just how Python can insure that we get O(1) complexity for the append operation. 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

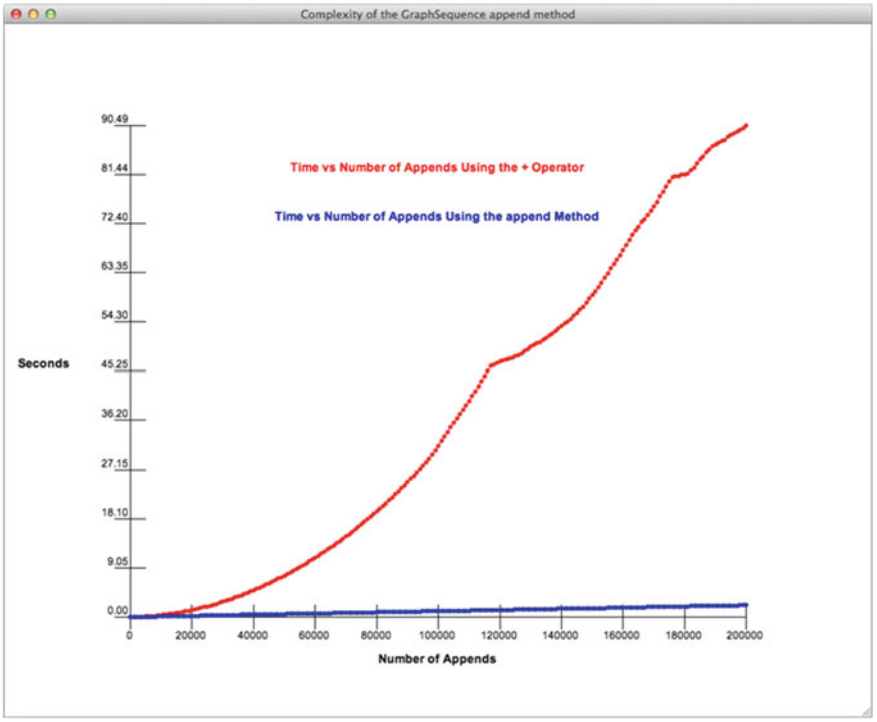

Fig. 2.4 The Complexity of Appending to a Pylist

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 Sect. 2.7.1.

2.7.1 Efficient Append

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

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

    ...
```

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

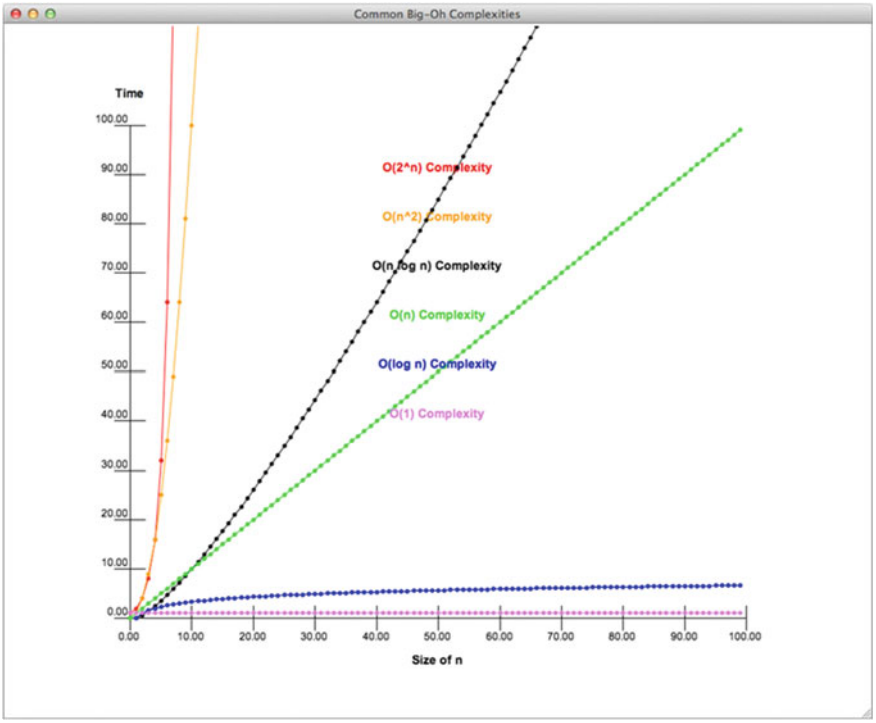

Fig. 2.5 Common Big-Oh Complexities

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-Oh 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-Oh notation definition presented earlier in this chapter.

2.9.1 Big-Oh 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-oh g of n. The definition of Big-Oh 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-Oh 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-Oh 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 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 $g(\mathrm{n}) = \mathrm{n}^2$ and the lower bound purple line is a plot of $g(\mathrm{n})/110$. 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 $c*g(\mathrm{n})$ 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.

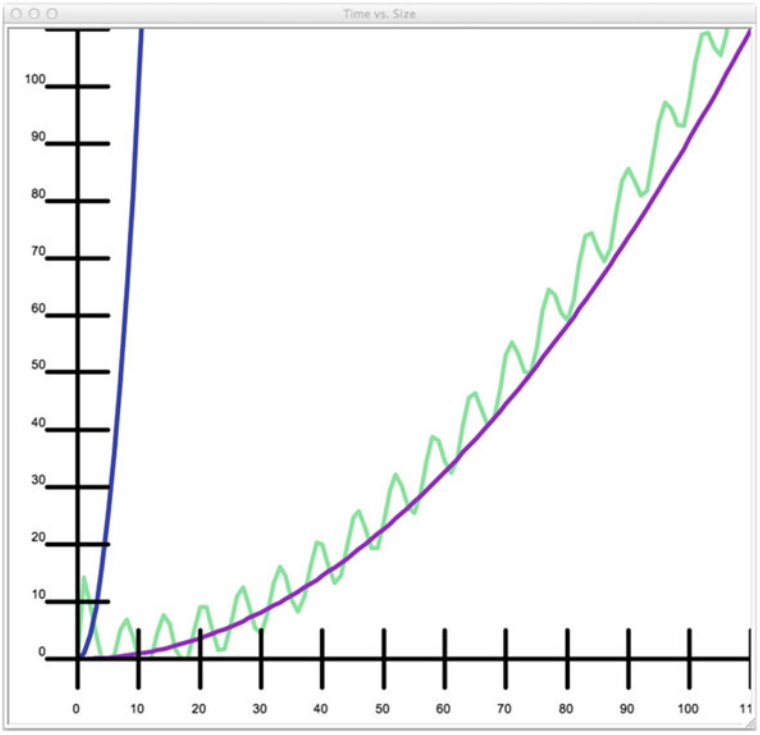

Fig. 2.6 A Lower and Upper Bound

Consider the PyList append operation discussed earlier in this chapter. The latest version of the PyList append method simply calls the Python append operation on lists. Python is implemented in C. It turns out that while Python supports an append operation for lists, lists are implemented as arrays in C and it is not possible to add to an array in C. An array can be allocated with a fixed size, but cannot have its size increased once created.

Pretend for a moment that Python lists, like C arrays, did not support the *append* method on lists and that the only way to create a list was to write something like *[None]*n* where *n* was a fixed value. Writing *[None]*n* creates a fixed size list of *n* elements each referencing the value *None*. This is the way C and C++ arrays are allocated. In our example, since we are pretending that Python does not support append we must implement our PyList append method differently. We can't use the *append* method and earlier in this chapter we saw that that adding on item at a time with the + operator was a bad idea. We'll do something a little different. Our PyList append operation, when it runs out of space in the fixed size list, will double the size of the list copying all items from the old list to the new list as shown in the code in Sect. 2.10.1.

2.10.1 A PyList Class

```
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()
```

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*.

2.10.2 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.

Consider a sequence of n append operations on an initially empty list. Appending the first element to the list is done in O(1) time since there is space for the first item added to the list because one slot was initially allocated in the list. Storing a value in an already allocated slot takes O(1) time. However, according to the accounting method, we'll claim that the cost of doing the append operation requires an additional two cyber dollars. This is still O(1) complexity. Each time we run out of space we'll double the number of slots in the fixed size list. Allocating a fixed size list is a O(1) operation regardless of the list size. The extra work comes when copying the elements from the old list to the new list.

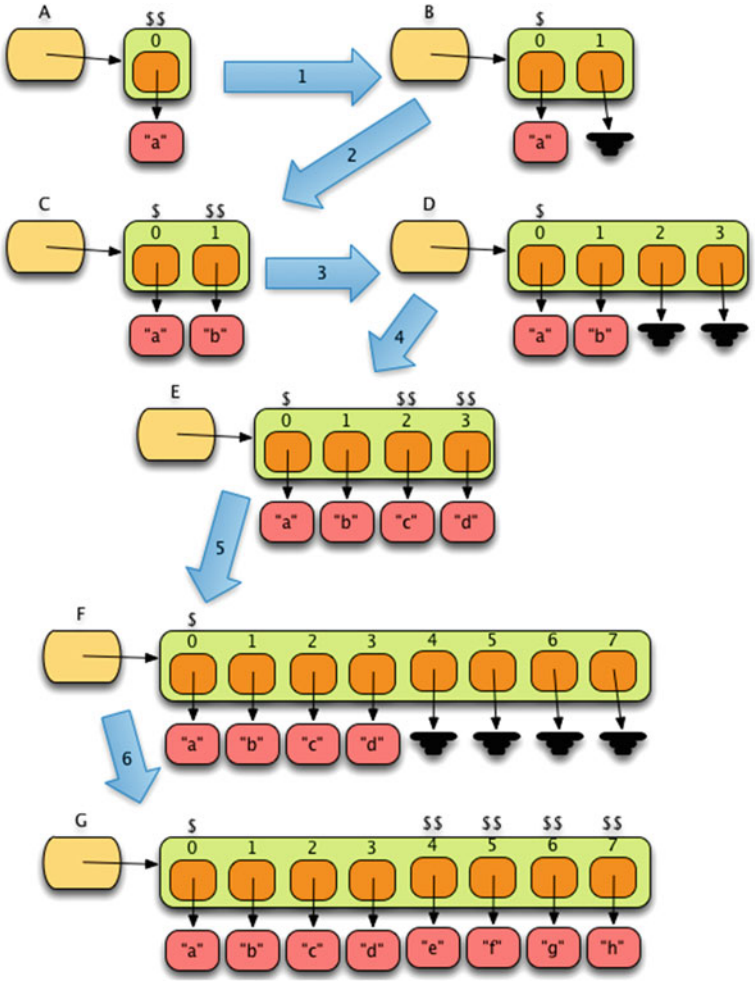

Fig. 2.7 Append Cyber Dollars

The first time we need to double the size is when the second append is called. There are two cyber dollars stored up at this point in time. One of them is needed when copying the one element stored in the old list to the new fixed size list capable of holding two elements. Transition one in Fig. 2.7 shows the two stored cyber dollars and the result after copying to the new list when moving from step A to step B.

When append is called on version B of the list the result is version C. At this point, three cyber dollars are stored to be used when doubling the list size to four locations. The first two are filled with the old contents of the list. Two of the three stored cyber dollars are used while copying these values to the new list. When the list of size four fills, two additional append operations have occurred, storing five cyber dollars. Four of these cyber dollars are used in the copy from step E to step F. Again, when the list of size eight fills in step G there are nine stored cyber dollars to be used in doubling the list size and copying the elements over.

But, what if we didn't double the size of the list each time. If we increased the list size by one half its previous size each time, we could still make this argument work if we stored four cyber dollars for each append operation. In fact, as long as the size of the list grows proportionally to its current size each time it is expanded this argument still works to prove that appending to a list is a O(1) operation when lists must be allocated with a fixed size.

As mentioned earlier, the Python list object is implemented in *C*. While Python provides an append operation, the *C* language can only allocate fixed size lists, called arrays in *C*. And yet, Python list objects can append objects in O(1) time as can be observed by experimentation or by analyzing the *C* code that implements Python list objects. The Python list *append* implementation achieves this by increasing the list size as described in this section when the fixed size array runs out of space to achieve an amortized complexity of O(1).

2.11 Chapter Summary

This chapter 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-Oh 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 is more efficient in terms of computational complexity? HINT: You need to be careful to consider the average case for this problem, not just a trivial case.
5. Write a short function that given a list, adds together all the values in the list and returns the sum. Write your program so it does this operation with varying sizes of lists. Record the time it takes to find the sum for various list sizes. Record this information in an XML file in the PlotData.py format. What complexity is this algorithm? Answer this in a comment at the top of your program and verify it with your experimental data. Compare this data with the built-in *sum* function in Python that does the same thing. Which is more efficient in terms of computational complexity? HINT: You need to be careful to consider the average case for this problem, not just a trivial case.
6. Assume that you have a datatype called the Clearable type. This data type has a fixed size list inside it when it is created. So Clearable(10) would create a clearable list of size 10. Objects of the Clearable type should support an append operation and a lookup operation. The lookup operation is called __getitem__(item). If *cl* is

a Clearable list, then writing *cl*[*item*] will return the item if it is in the list and return *None* otherwise. Writing *cl*[*item*] results in a method call of cl.__getitem__(item). Unlike the append operation described in Sect. 2.10.1, when the Clearable object fills up the list is automatically cleared or emptied on the next call to append by setting all elements of the list back to *None*. The Clearable object should always keep track of the number of values currently stored in the object. Form a theory about the complexity of the append operation on this datatype. Then write a test program to test the Clearable object on different initial sizes and numbers of append operations. Create one sequence for each different initial size of the Clearable datatype and write your results in the plot format described in this chapter. Then comment on how your theory holds up or does not hold up given your experimentation results.