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.

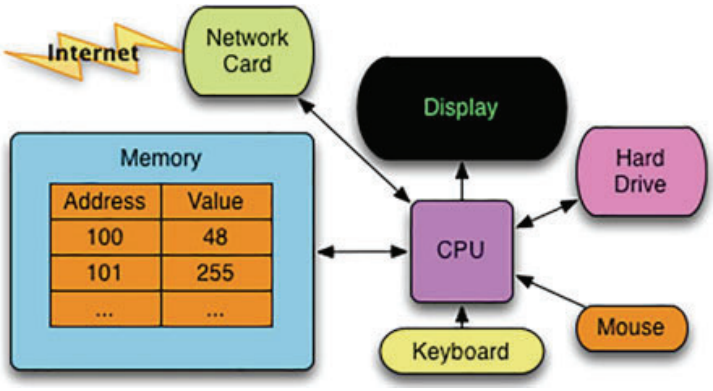

Fig. 2.1 Conceptual View of a Computer

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 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 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
```

```
    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')

```

```
87      file.write('  </Sequence>\n')
88      file.write('</Plot>\n')
89      file.close()
90
91  if __name__ == "__main__":
92      main()
```

Listing 2.1 List Access Timing

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 Listing 2.1 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 Listing 2.2. The data is abbreviated, but the format is as shown in Listing 2.2.

```
1   <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2   <Plot title="Average List Element Access Time">
3       <Axes>
4       <XAxis min="1000" max="200000">List Size</XAxis>
5       <YAxis min="20.244" max="60">Microseconds</YAxis>
6       </Axes>
7       <Sequence title="Average Access Time vs List Size" color="red">
8       <DataPoint x="1000" y="33.069"/>
9       <DataPoint x="2000" y="27.842"/>
10      <DataPoint x="3000" y="23.908"/>
11      <DataPoint x="4000" y="26.349"/>
12      <DataPoint x="5000" y="23.212"/>
13      <DataPoint x="6000" y="23.765"/>
14      <DataPoint x="7000" y="21.251"/>
15      <DataPoint x="8000" y="21.321"/>
16      <DataPoint x="9000" y="23.197"/>
17      <DataPoint x="10000" y="21.527"/>
18      <DataPoint x="11000" y="35.799"/>
19      <DataPoint x="12000" y="22.173"/>
20      ...
21      <DataPoint x="197000" y="26.245"/>
22      <DataPoint x="198000" y="30.013"/>
23      <DataPoint x="199000" y="25.888"/>
24      <DataPoint x="200000" y="23.578"/>
25      </Sequence>
26      <Sequence title="Access Time Distribution" color="blue">
27      <DataPoint x="219" y="41.0"/>
28      <DataPoint x="2839" y="38.0"/>
29      <DataPoint x="5902" y="38.0"/>
30      <DataPoint x="8531" y="58.0"/>
31      <DataPoint x="11491" y="38.0"/>
32      <DataPoint x="15415" y="38.0"/>
```

```
33      <DataPoint x="17645" y="31.0"/>
34      <DataPoint x="18658" y="38.0"/>
35      <DataPoint x="20266" y="40.0"/>
36      <DataPoint x="21854" y="31.0"/>
37      ...
38      <DataPoint x="197159" y="37.0"/>
39      <DataPoint x="199601" y="40.0"/>
40      </Sequence>
41  </Plot>
```

Listing 2.2 A Plot XML Sample

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 Listing 2.2 and plot the sequences to the screen. The PlotData.py program is given in Sect. 22.4.

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 1000 element accesses on a list of the given size. The average access time (computed from a sample of 1000 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

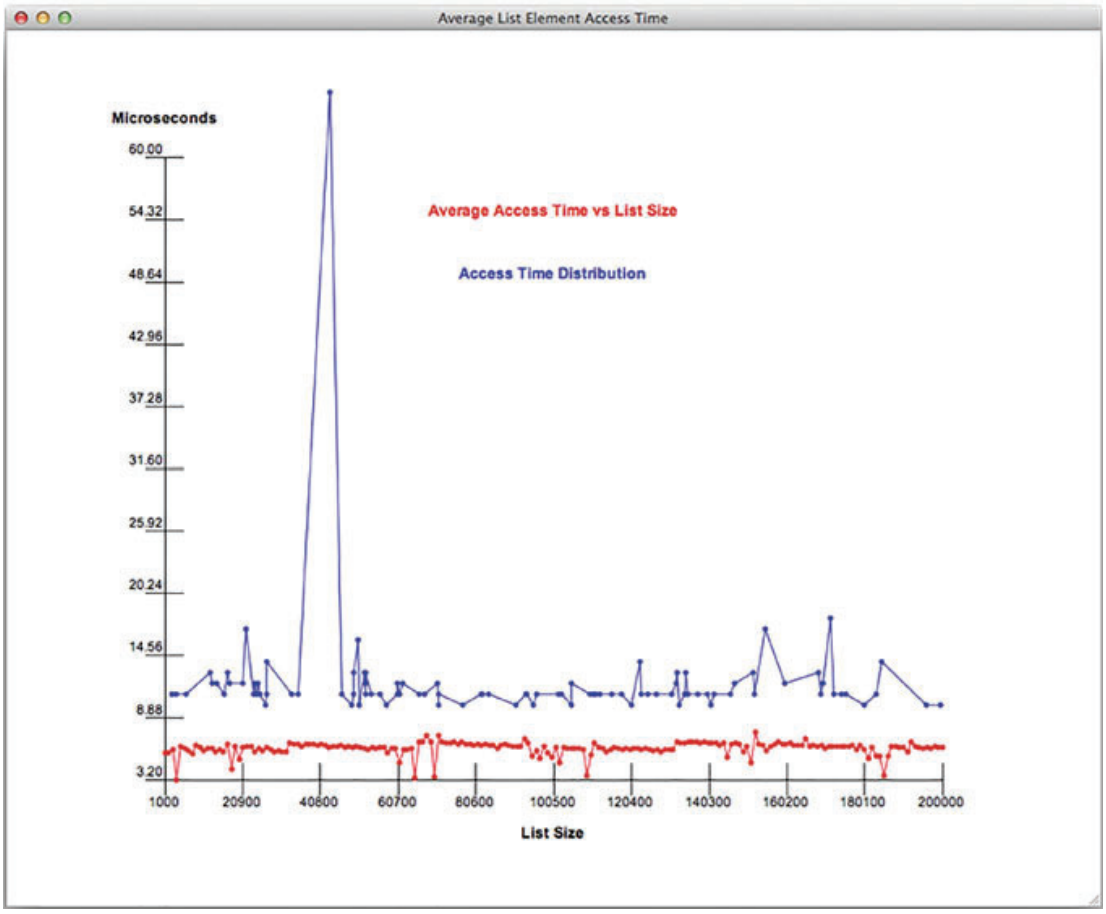

Fig. 2.2 Access times in a Python list

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 $f(n)$.

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). Because the time to access an element does not depend on n, we can pick g(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 μ s 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 $g(n) = 1$ is arbitrary in computing the complexity of accessing an element of a list. We could have chosen $g(n) = 2$. If $g(n) = 2$ 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.

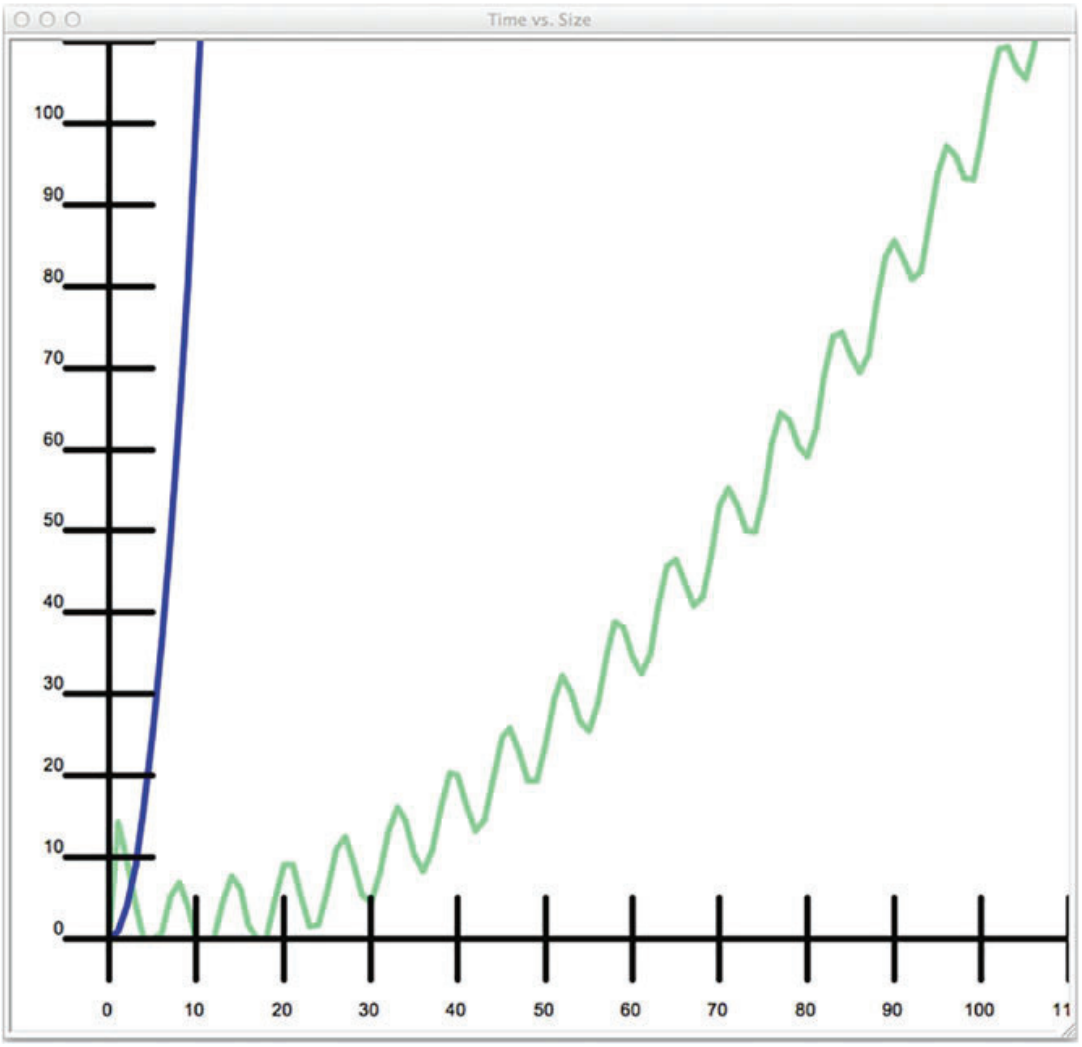

Fig. 2.3 An Upper Bound

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