CAB301 Assignment 1

Empirical Analysis of an Algorithm for Inserting a Number Into a Set

Student name:
(Student no. n1234567)

Date submitted: 15 April 2018

Summary

This report summarises the outcomes of several experiments conducted to measure the time complexity of an algorithm for inserting a number into a set, where sets are represented as ordered linked lists. The algorithm was implemented as a Python program and both the number of basic operations performed by the program and its execution time were measured. The experimental results were found to be consistent with the theoretical predictions for this algorithm.

1 Description of the Algorithm

The algorithm of interest inserts a number into a set, assuming that sets are represented as ordered linked lists [1, §2.3.3]. Insertions must preserve the list's ordering and duplicate numbers should not be inserted. Using ordered lists to represent sets, instead of unordered ones, reduces the average time required to insert a new number [1, Ex. 2.61].

The algorithm required to do this is a common textbook example. Berman and Paul illustrate how insertion into an ordered list works with an example [2, pp. 44–45], although they leave the algorithm itself "as an exercise" for the reader, as do Abelson and Sussman [1, Ex. 2.61]. The particular version of the algorithm used in this experiment is shown in Figure 1 on page 7. It is based on Schneider *et al.*'s explanation of how to insert an item at a particular point in a linked list [6, pp. 247–248].

As usual in linked list algorithms, insertion of a new node at the beginning of the list (condition b in Figure 1) needs to be treated as a special case (statements c and d). Otherwise, however, the algorithm uses a loop (statements f and g) to search for the place to insert the new node. The loop stops either when the end of the list is reached, or when a number larger than the one we need to insert is encountered. Since we don't want to insert duplicate numbers, a new node is then inserted into the list (statements i and j) only if the current node's contents are not the same as the number to be inserted (condition h).

2 Theoretical Analysis of the Algorithm

This section describes the algorithm's anticipated time complexity from a theoretical perspective.

2.1 Identifying the Algorithm's Basic Operation

Clearly the aspect of the algorithm in Figure 1 that has the greatest influence on its execution time is searching for the place in the list to insert the new number. Therefore, a reasonable choice for the algorithm's basic operation is any test used to see whether or not the new number can be inserted at the 'current' place in the list. In Figure 1 this is done in the outermost '**if**' statement (condition b) and in the '**while**' loop (condition f).

For simplicity, we treated these two Boolean expressions as atomic, although a more precise analysis would be possible by noting that the second disjunct in the first expression, and the second conjunct in the second expression, will not be evaluated if the end of the list is reached [4, Ex. 2.6(1)]. Nevertheless, this difference is insignificant for long lists. Similarly, we chose not to count the test in the innermost '**if**' statement (condition h) because it is *always* performed for non-empty lists, and is thus also insignificant in the complexity analysis.

2.2 Best-Case Efficiency

The best-case scenario for this algorithm is when the new number can be added at the beginning of the list, either because the set is empty or because the number at the head of the list is bigger than the one to be added. In either of these situations only one basic operation (in the outermost '**if**' statement) must be performed. In Levitin's notation [4, p. 48] we say $C_{best}(n) = 1$, where n denotes the length of the list before the insertion is performed.

2.3 Worst-Case Efficiency

The worst-case scenario is where the new number must be added after the last one in the list, meaning that we are forced to inspect all n numbers in the set before we can insert the new one. Counting the final evaluation of the '**while**' test, which recognises that we have searched beyond the end of the list, we have $C_{worst}(n) = n + 1$.

2.4 Average-Case Efficiency

Although we could not find a description of this specific algorithm's average-case efficiency in the literature, it is clear that the problem of finding a particular place in an ordered linked list is analogous to searching sequentially for a particular item in an array. Levitin analyses the behaviour of just such a algorithm [4, pp. 47–49], as do Berman and Paul [2, p. 34]. They conclude that the average number of basic operations required to find an item in an array of length n is $(n + 1)/2$. Therefore, we similarly assume that $C_{avg}(n) = (n + 1)/2$ for the set insertion algorithm.

2.5 Aggregate Efficiency

After each insertion the size of the set grows, so the average time required for consecutive insertions into the same linked list increases. For this kind of algorithm, therefore, we are interested in the aggregate (or 'cumulative') efficiency of inserting n items, for various values of n.

Using the average-case equation for individual insertions from Section 2.4, the average time required to create a set by inserting n distinct numbers should be:

$$\begin{aligned} & \sum_{i=1}^{n} \frac{i+1}{2} \\ = \quad & \text{by arithmetic} \end{aligned}$$

$$
\begin{aligned}
& \frac{n + \sum_{i=1}^{n} i}{2} \\
= \quad & \text{by sum of a series [4, p. 470][2, p. 865]} \\
& \frac{n + \frac{n \cdot (n+1)}{2}}{2} \\
= \quad & \text{by arithmetic} \\
& \frac{n^2 + 3n}{4}
\end{aligned}
$$

This equation tells us how many basic operations should be required to create a set of size n on average (assuming that each number inserted is distinct). For instance, creating a set of size 7500 should take around 14,068,125 basic operations, and creating one of size 10000 should take around 25,007,500 basic operations.

2.6 Order of Growth

Abelson and Sussman briefly note that finding a particular element in a sorted linked list of length n is of order $\Theta(n)$ [1, p. 154], and the equations above confirm that the dominant factor for inserting a single item into a set is the set's size, i.e., the length of the corresponding ordered linked list. Thus, when measuring how long it takes to insert a single item we should see linear ('straight line') growth with the size of the set.

The equation in Section 2.5 above for the aggregate efficiency of creating a whole set of size n suggests that set creation is of efficiency class $\Theta(n^2)$. Thus, when measuring how long it takes to create complete sets we expect to see quadratic ('convex') growth with the total size of the set.

3 Methodology, Tools and Techniques

This section briefly summarises the computing environment used for the experiments.

1. The algorithm and the experiments were implemented in the Python programming language. Python is freely available as open source software and has an exceptionally clear and simple syntax [5, 3]. Downey *et al.*'s book was used as a guide for creating linked list nodes as Python objects [3, Ch. 17].

2. The experiments were performed on an Apple Macintosh PowerBook G4 laptop computer, running the UNIX-based Mac OS X operating system. Python's pseudorandom number generator [5, p. 301] was used to produce test data and Python's time module [5, p. 246] was used to measure execution times. Since some experiments required the numbers inserted into the set to be distinct (so that the set's size keeps increasing) care was taken to confirm that the random number generator produced distinct numbers for the tests conducted. For experiments involving execution time measurements, the number of other software applications running concurrently with the tests was minimised.

3. Graphs of the experimental results were produced using Apple's *Grapher* utility. This program can import sets of data points from a text file, so the test programs were designed to write their results to files in a format that could be imported directly into Grapher. Figures 2 to 6 below were all prepared in Grapher, and exported in Portable Document Format. This report was then prepared using the LaTeX typesetting system.

4 Experimental Results

This section describes the outcomes of the experiments and compares the results with the theoretical predictions from Section 2. The programming language implementation of the

algorithm from Figure 1 is shown in Appendix A.

4.1 Functional Testing

To test the functional correctness of the program in Appendix A, the small test program described in Appendix B was used. The test program creates an empty set, inserts a given sequence of numbers, and displays the set's final value. Several different tests were performed for various typical and 'extreme' cases.

For instance, in the particular test shown in Appendix B, the numbers 0, 4, 5, 3, 2, 8, 9, 2, 7, 6, 1 and 5 are inserted into the set, in that order. This test produced the following output, confirming that the numbers in the resulting linked list are ordered correctly and that duplicate numbers are ignored.

```
The set is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
The set contains 10 items
```

In another test, the numbers 5, 4, 3, 2 and 1 were inserted into the set, in that order. The output in this situation was as follows, confirming that the program can handle numbers provided in reverse order.

```
The set is {1, 2, 3, 4, 5}
The set contains 5 items
```

At the other extreme, a test was performed in which the numbers were already in sequence. Inserting numbers 6, 7, 8, 9, 10, 11 and 12 produced the following correct output.

```
The set is {6, 7, 8, 9, 10, 11, 12}
The set contains 7 items
```

4.2 Best and Worst-Case Number of Basic Operations

To measure the algorithm's best and worst-case efficiency, the number of basic operations required to insert individual numbers into a set were counted using the program shown in Appendix C. It creates a set of a given size (10000 in this case) by inserting distinct, randomly-generated numbers one at a time. The number of basic operations required for every 100th insertion was recorded. The results are shown in Figure 2 on page 8.

This graph confirms that the number of basic operations is bounded from both below and above as predicted in Sections 2.2 and 2.3, respectively. Even when the set is quite large some insertions occur near the beginning of the linked list and take only a few basic operations. Conversely, no insertion ever requires more basic operations than the size of the set. For the particular experiment shown in Figure 2, the best case stays constant at 1, even as the set becomes large. For instance, the 8500th insertion required only one basic operation. The worst case grows linearly with the size of the set. For instance, the 9200th insertion required 8833 basic operations.

4.3 Average-Case Number of Basic Operations

To measure the algorithm's average-case behaviour the test program in Appendix D was used. Again this program creates sets (of size 10000) by inserting random numbers, but in this case ten separate sets were created and the number of basic operations for insertions into sets of the same size was averaged. The results are shown in Figure 3 on page 9.

Although the graph is still fairly 'noisy', due to the randomness in the data, it is clear that the number of basic operations grows at half the rate of the set's size as predicted in Section 2.4. For instance, the average number of basic operations required for the 5200th insertion was 2594, and the average for the 9600th insertion was 4827.

4.4 Aggregate Number of Basic Operations

To measure the algorithm's aggregate efficiency the test program described in Appendix E was used. This created sets of several different sizes, in this case 10, 50, 100, 500, 1000, 2500, 5000, 7500 and 10000 items, and counted the total number of basic operations required to create each one. The results are shown in Figure 4 on page 10.

The graph displays a convex shape consistent with the efficiency-class prediction in Section 2.6. Also, the measured number of basic operations accords well with the formula for the average number of basic operations in Section 2.5. For instance, the theoretical prediction says that it should take 14,068,125 operations to create a set of size 7500 on average. The actual measured value for creating a set of this size, using random data, was 14,021,443. For 10000 insertions the theory predicts 25,007,500 basic operations, and the measured value was 24,906,748.

4.5 Average-Case Execution Time

Our first attempt to measure the execution time of individual set insertions was unsuccessful. Even for large sets, each insertion occurred too quickly to measure accurately. Therefore, it was decided to measure the execution time of groups of consecutive insertions.

The program shown in Appendix F does this by measuring how long it takes to insert groups of 100 items into a set. This strategy proved very effective as shown by the experimental results in Figure 5 on page 11. (Measuring groups of insertions also had the beneficial side-effect of 'smoothing' the randomness in the test data.) The graph exhibits a clear linear ('straight line') growth as was predicted in Section 2.6. This can be confirmed by noting that the ratios of the total number of items inserted versus the corresponding execution time remain comparable. Some examples from the experiment shown in Figure 5 are $2500/0.73 = 3425$, $5100/1.62 = 3148$ and $9600/2.95 = 3254$.

4.6 Aggregate Execution Time

Appendix G shows the code for measuring the aggregate execution time required to create sets of different sizes. The experimental results are shown in Figure 6 on page 12. They confirm the convex shape of the curve anticipated in Section 2.6.

Overall, the execution time results are consistent with the measured number of basic operations. For instance, the ratios of the square of the size of the set versus the execution time are similar. For the largest three values in Figure 6 we obtain $5000^2/35.6 = 702247$, $7500^2/81.08 = 693759$ and $10000^2/146.56 = 682314$. We can therefore conclude that no significant memory management or operating system overheads impacted on the execution times for the sizes of set considered. (However, this does not guarantee that such effects would not be seen for much larger sets.)

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, second edition, 1996. ISBN 0-262-51087-1.

[2] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel and Distributed*. Thomson, 2005. ISBN 0-534-42057-5.

[3] A. Downey, J. Elkner, and C. Meyers. *How to Think Like a Computer Scientist: Learning with Python*. Green Tea Press, 2002. http://www.greenteapress.com/thinkpython/.

[4] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, second edition, 2007. ISBN 0-321-36413-9.

[5] A. Martelli. *Python in a Nutshell*. O'Reilly, 2003. ISBN 0-596-00188-6.

[6] G. M. Schneider, S. W. Weingart, and D. M. Perlman. *An Introduction to Programming and Problem Solving With Pascal*. John Wiley and Sons, 1978. ISBN 0-471-04431-8.

```
a.  ALGORITHM SetInsert(newNum)
      // Inserts number newNum into the sorted linked list pointed to by global
      // variable firstNode, provided that the number is not already in the list,
      // and preserving the list's ordering
b.    if firstNode = Null or newNum < firstNode↑Contents
c.      newNode ← Node(newNum, firstNode)
d.      firstNode ← newNode
      else
e.      currNode ← firstNode
f.      while currNode↑Next ≠ Null and
                currNode↑Next↑Contents ≤ newNum do
g.        currNode ← currNode↑Next
h.      if currNode↑Contents ≠ newNum
i.        newNode ← Node(newNum, currNode↑Next)
j.        currNode↑Next ← newNode
```

Figure 1: The algorithm to be analysed. Let 'Null' denote the null pointer. Assume that constructor 'Node' returns a linked-list node with fields 'Contents' and 'Next'. Let '$p\uparrow f$' denote field f of the node pointed to by pointer p.

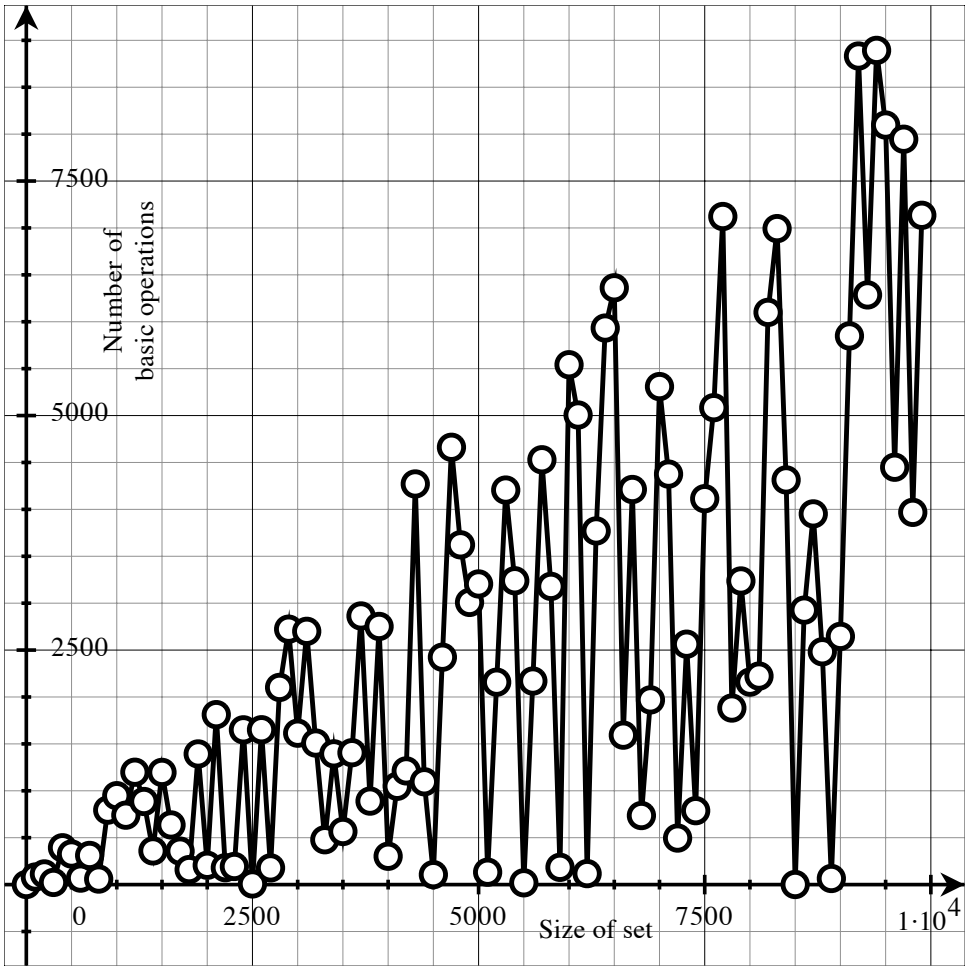

Figure 2: Best and worst-case efficiency. This graph shows the measured number of basic operations required to perform insertions into a set using random data (Appendix C). One hundred data points are shown, where each point represents the number of basic operations required to insert a particular item into the set. The smallest and largest values seen as the size of the set grows indicate that the best and worst cases are constant and linear, respectively.

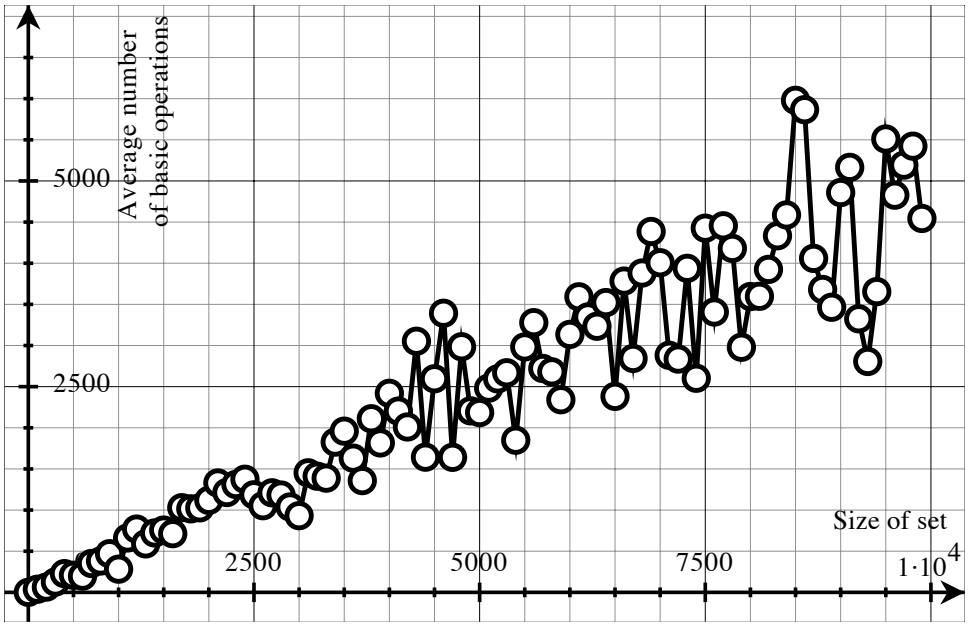

Figure 3: Average number of basic operations. This graph shows the measured number of basic operations required to perform insertions into a set using random data (Appendix D). One hundred data points are shown, where each point represents the average of 10 tests. The result confirms that the average number of basic operations required to insert an item into the list grows linearly as half the size of the set.

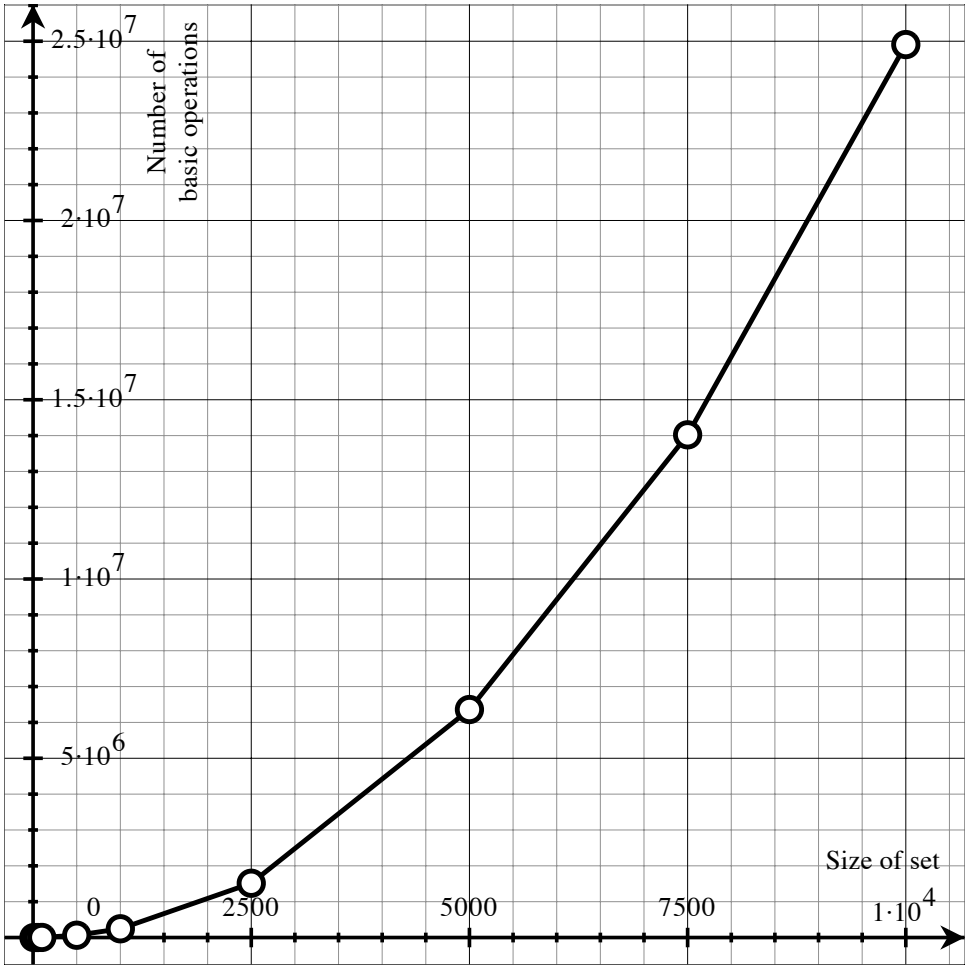

Figure 4: Aggregate number of basic operations. This graph shows the measured number of basic operations required for creating sets of different sizes (Appendix E). Nine data points are shown, each representing the total number of basic operations required to create a set of a given size (10, 50, 100, 500, 1000, 2500, 5000, 7500 and 10000 items, respectively). The result confirms that the total number of insertions grows quadratically with the size of the set.

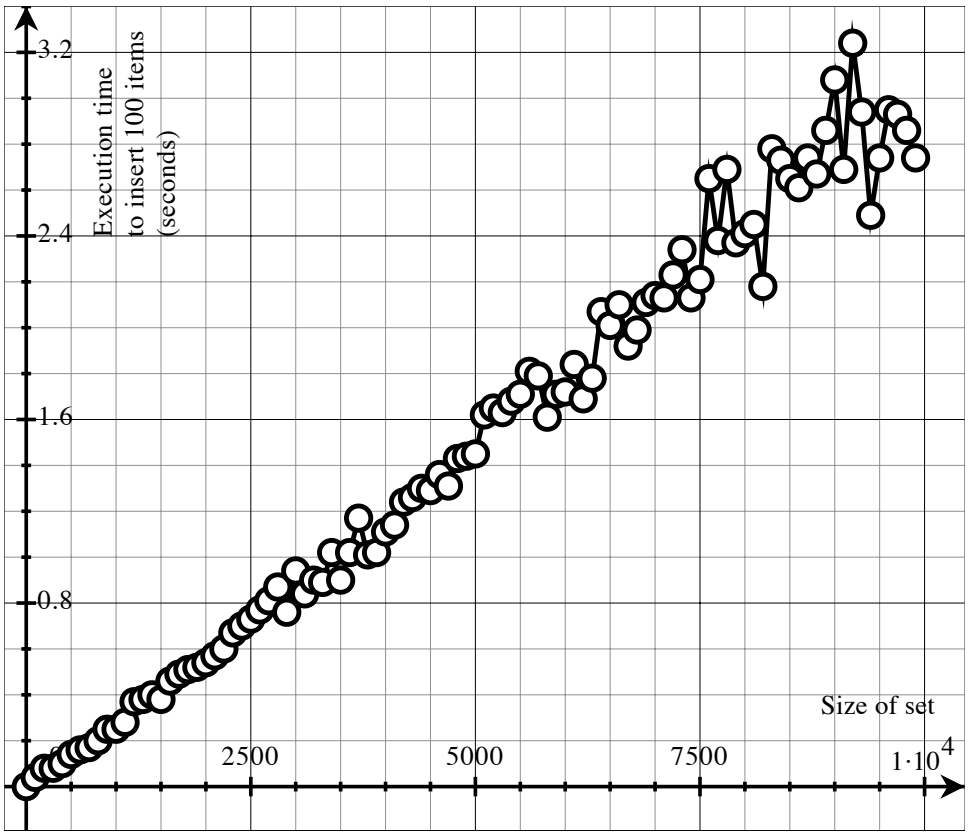

Figure 5: Execution time for inserting items into a set. This graph shows the measured execution times for inserting groups of items into a set (Appendix F). One hundred data points are shown, where each point represents the elapsed time required to insert 100 items into the set. The result confirms that the execution time grows linearly with the size of the set.

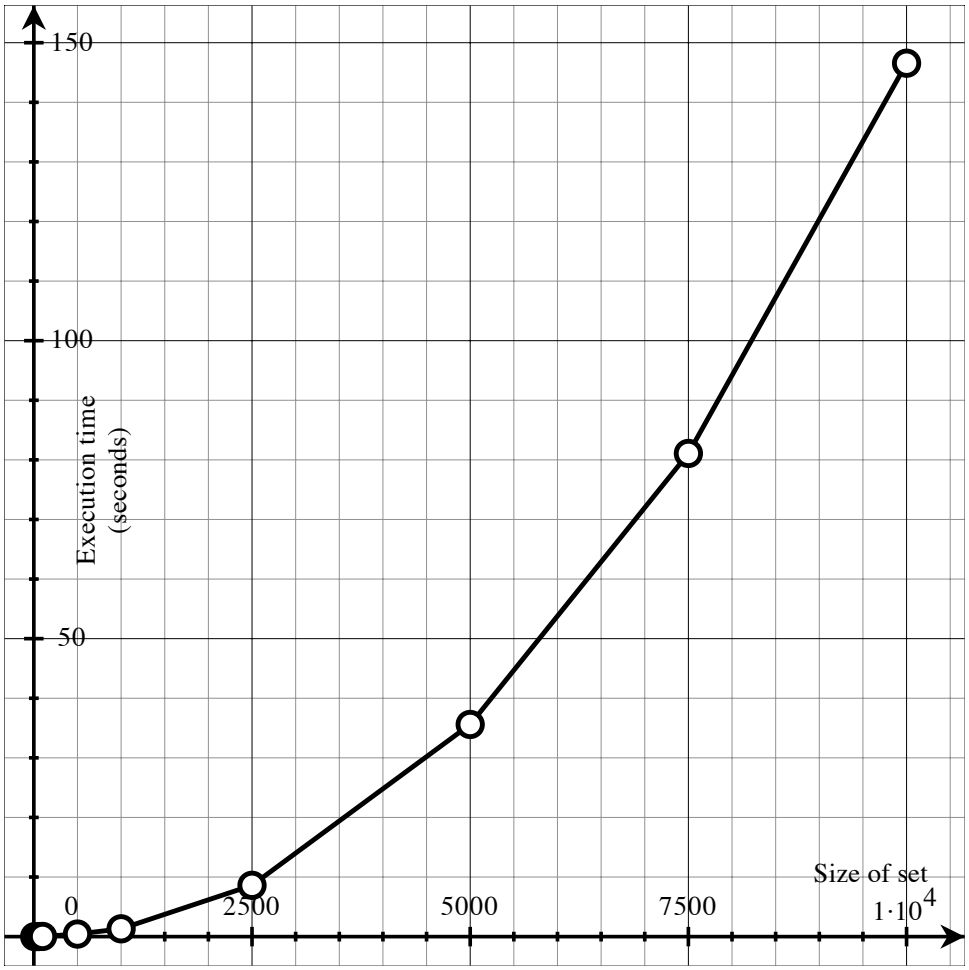

Figure 6: Aggregate execution time. This graph shows the measured total execution times for creating sets of different sizes (Appendix G). Nine data points are shown, where each data point represents the total elapsed execution time for creating a set of a particular size (10, 50, 100, 500, 1000, 2500, 5000, 7500 and 10000 items, respectively). The results suggest that the execution time has a quadratic growth, as was predicted for the number of basic operations.

Appendix A Code for the Algorithm

This appendix presents the Python code written to implement the algorithm in Figure 1. Since Python does not have linked lists as a built-in data structure, Downey *et al.*'s implementation of linked lists in Python was adapted [3, Ch. 17]. They explain how to create linked lists nodes as objects that include the next node as an attribute.

The program comprises two classes. The first, shown below, implements linked list nodes as objects which have two attributes, the node's contents (`Contents`) and the next node in the list (`Next`). Python's 'null' value `None` is used as the null pointer. In Python the special method `init` is invoked when an object is created and special method `str` is invoked when the 'string' version of an object is required (e.g., in a `print` statement). In our case the `init` method (lines 3–5) just stores the node's contents and next 'pointer' as local attributes, and the `str` method (lines 7–8) just returns the string representation of the node's contents.

```
1. class Node:
2.
3.   def __init__(self, initContents=None, initNext=None):
4.     self.Contents = initContents
5.     self.Next = initNext
6.
7.   def __str__(self):
8.     return str(self.Contents)
```

The code for the linked list class used to represent sets of numbers is shown below. Here a set has two attributes, a pointer to the first node in the linked list holding the set's contents (`firstNode`), and the set's size (`setSize`). (The algorithm in Figure 1 does not require us to maintain the set's size, but this attribute proved useful when checking to ensure that the random number generator was producing distinct numbers.)

When a set is created it has no contents and is of size zero (lines 3–5). The method for returning the contents of a set as a string (lines 23–32) is based on some similar code for printing linked lists described by Downey *et al* [3, pp. 181, 186]. It was used when testing the algorithm's correctness.

Most importantly, the `SetInsert` method (lines 7–21) implements the algorithm from Figure 1 on page 7. There is a direct correspondence between the algorithm and the program code; the letters after the line numbers refer to statements in the algorithm. The only significant differences are the additional code to maintain the set's size (lines 12 and 21), and the object-oriented '`self`' prefixes.

```
 1. class Set:
 2.
 3.   def __init__(self):
 4.     self.setSize = 0
 5.     self.firstNode = None
 6.
7(a).   def SetInsert(self, newNum):
8(b).     if self.firstNode == None or
 9.          newNum < self.firstNode.Contents:
10(c).       newNode = Node(newNum, self.firstNode)
11(d).       self.firstNode = newNode
 12.         self.setSize = self.setSize + 1
 13.    else:
14(e).       currNode = self.firstNode
15(f).       while currNode.Next != None and
 16.               currNode.Next.Contents <= newNum:
```

```
17(g).          currNode = currNode.Next
18(h).        if currNode.Contents != newNum:
19(i).          newNode = Node(newNum, currNode.Next)
20(j).          currNode.Next = newNode
  21.           self.setSize = self.setSize + 1
  22.
  23.   def __str__(self):
  24.     def PrintSetItems(currNode):
  25.       stringRep = ""
  26.       while currNode != None:
  27.         stringRep = stringRep + str(currNode.Contents)
  28.         currNode = currNode.Next
  29.         if currNode != None:
  30.           stringRep = stringRep + ", "
  31.       return stringRep
  32.     return "{" + PrintSetItems(self.firstNode) + "}"
```

Appendix B Code for Functional Testing

The following code was used to test the functional correctness of the algorithm's implementation. This is the program that produced the output described in Section 4.1.

```
1. setOfNumbers = Set()
2.
3. for setItem in [0, 4, 5, 3, 2, 8, 9, 2, 7, 6, 1, 5]:
4.   setOfNumbers.SetInsert(setItem)
5.
6. print "The set is", setOfNumbers
7. print "The set contains ", setOfNumbers.setSize,
8.       " items"
```

The program creates an empty set (line 1), adds several numbers to it in the given order (lines 3–4), and then displays the resulting set (line 6) and its size (lines 7–8). This made it possible to check that the list is correctly ordered and that duplicate numbers appear only once in the set. Different tests were performed by changing the sequence of numbers to be inserted (line 3).

Appendix C Code for Counting the Number of Basic Operations

To measure the number of basic operations (Section 2.1) performed by the algorithm it was first necessary to modify procedure `SetInsert` so that it increments a counter whenever a basic operation is executed. The following version of the `SetInsert` procedure from Appendix A was written to do this. New statements are underlined.

```
 7. def SetInsertCount(self, newNum):
7′.   numBasic = 0
 8.   if self.firstNode == None or
 9.      newNum < self.firstNode.Contents:
9′.     numBasic = numBasic + 1
        ...
15.     while currNode.Next != None and
16.           currNode.Next.Contents <= newNum:
16′.      numBasic = numBasic + 1
17.       currNode = currNode.Next
17′.    numBasic = numBasic + 1
18.     if currNode.Contents != newNum
          ...
21.       self.setSize = self.setSize + 1
21′.  return numBasic
```

New variable `numBasic` is initialised as zero (line 7′), is incremented each time one of the basic operations identified in Section 2.1 is performed (lines 9′, 16′ and 17′), and is returned when the method terminates (line 21′). Note that the increment on line 17′ counts the last time the '**while**' loop's condition is evaluated and is found to be false.

Using this 'instrumented' version of the `SetInsert` procedure, the following code was then used to count the number of basic operations performed when creating a set of a certain size. Every 100th insertion is recorded. This is the program that was used to produce the results shown in Figure 2 on page 8.

```
 1. setSize = 10000
 2. sampleSep = 100
 3.
 4. resultsfile = open('insertop.txt', mode='w')
 5.
 6. testSet = Set()
 7.
 8. for itemNum in xrange(setSize):
 9.   opCount = testSet.SetInsertCount(random.random())
10.   if (itemNum % sampleSep) == 0:
11.     resultsfile.write(str(itemNum) + " " +
12.                       str(opCount) + " 0\n")
13.
14. resultsfile.close()
```

The program opens a file to store the results of the experiment (line 4), creates an empty set (line 6), inserts the required quantity of random numbers into the set (lines 8–9), records the number of basic operations performed for every 100th insertion (lines 10–12), and closes the results file (line 14). The '`%`' operator is the remainder of integer division in Python [5, p. 43], and is used on line 10 to check whether the number of insertions performed so far is a whole multiple of 100.

Appendix D Code for Averaging the Number of Basic Operations

The following program was used to count the average number of basic operations needed to insert numbers into sets of different sizes. This was the program used to produce the results shown in Figure 3 on page 9.

```
1. setSize = 10000
2. sampleSep = 100
3. numTests = 10
4.
5. resultsfile = open('opaverage.txt', mode='w')
6.
7. testSet = []
8. for setNum in xrange(numTests):
9.   testSet = testSet + [Set()]
10.
11. for itemNum in xrange(setSize):
12.   opCount = 0
13.   for setNum in xrange(numTests):
14.     opCount =
15.       testSet[setNum].SetInsertCount(random.random()) +
16.       opCount
17.   if (itemNum % sampleSep) == 0:
18.     resultsfile.write(str(itemNum) + " " +
19.                       str(opCount // numTests) +
20.                       " 0\n")
21.
22. resultsfile.close()
```

The program is similar to the one in Appendix C and relies on the same instrumented version of procedure `SetInsert` (line 15). Rather than creating a single set, however, it creates an array of sets (lines 7–9), inserts an item into each set in the array at each iteration, counting the total number of basic operations for all the insertions (lines 13–16), and then prints the average number of basic operations to the results file (lines 17–20). The '`//`' operator on line 19 is integer (truncating) division in Python [5, p. 43].

Appendix E Code for Aggregating the Number of Basic Operations

The following program was used to measure the number of basic operations required to create complete sets of different sizes. This was the program used to produce the results shown in Figure 4 on page 10.

```
1. setSizes = [10, 50, 100, 500, 1000, 2500, 5000,
2.             7500, 10000]
3.
4. resultsfile = open('insertops.txt', mode='w')
5.
6. for numItems in setSizes:
7.   testSet = Set()
8.   opCount = 0
9.   for itemNum in xrange(numItems):
10.    opCount = opCount +
11.          testSet.SetInsertCount(random.random())
12.  resultsfile.write(str(numItems) + " " +
13.                    str(opCount) + " 0\n")
14.
15. resultsfile.close()
```

The program relies on the instrumented version of procedure `SetInsert` shown in Appendix C (line 11). For each given set size (lines 1–2 and 6) the program creates an empty set (line 7), sums the total number of basic operations required to insert the specified number of items into the set (lines 8–11), and prints the total (lines 12–13).

Appendix F Code for Measuring Execution Times for Individual Insertions

The following program was used to measure how long it takes to insert a randomly-generated number into a set using the algorithm in Figure 1. The program uses Python's `clock` procedure [5, pp. 246–247] to record the starting and finishing times for insertions and then calculates the difference between the two to determine the elapsed execution time. This is the program used to produce the results shown in Figure 5 on page 11.

```
 1. setSize = 10000
 2. sampleSep = 100
 3.
 4. resultsfile = open('inserttime.txt', mode='w')
 5.
 6. testSet = Set()
 7.
 8. StartTime = time.clock()
 9. for itemNum in xrange(setSize):
10.   if (itemNum % sampleSep) == 0:
11.     testSet.SetInsert(random.random())
12.     EndTime = time.clock()
13.     resultsfile.write(str(itemNum) + " " +
14.                       str(EndTime - StartTime) +
15.                       " 0\n")
16.     StartTime = time.clock()
17.   else:
18.     testSet.SetInsert(random.random())
19.
20. resultsfile.close()
```

This code relies on the original version of the `SetInsert` procedure from Appendix A (lines 11 and 18), rather than the instrumented version from Appendix C, because we don't want to include the overheads of incrementing the operation counter in our measurements.

As explained in Section 4.5, it was found necessary to measure the execution time of groups of insertions, since individual insertions were too quick to measure accurately. In the code above insertions are measured in groups of 100. The program records the starting time for each group (lines 8 and 16) and the corresponding finishing time (line 12), and then writes the difference between the two to the results file (lines 13–15).

Appendix G Code for Measuring Execution Times for Creating Entire Sets

The following program measures the elapsed time required to create entire sets of various sizes by inserting randomly-generated numbers. This is the program that was used to produce the results shown in Figure 6 on page 12.

```
1. setSizes = [10, 50, 100, 500, 1000, 2500, 5000,
2.             7500, 10000]
3.
4. resultsfile = open('inserttimes.txt', mode='w')
5.
6. for numItems in setSizes:
7.
8.   testSet = Set()
9.
10.  StartTime = time.clock()
11.  for itemNum in xrange(numItems):
12.    testSet.SetInsert(random.random())
13.  EndTime = time.clock()
14.
15.  resultsfile.write(str(numItems) + " " +
16.                     str(EndTime - StartTime) + " " +
17.                     str(testSet.setSize) + "\n")
18.
19. resultsfile.close()
```

This code relies on the original version of the `SetInsert` procedure from Appendix A (line 12), rather than the instrumented version from Appendix C, because we don't want to include the overheads of incrementing the operation counter in our measurements.

For each set in the given range of sizes (lines 1–2 and 6) the program creates an empty set (line 8), records the starting time (line 10), inserts the required number of values into the set (lines 11–12), records the finishing time (line 13), and prints the difference between the finishing and starting times (lines 15–17).