CHAPTER 4

Algorithm Analysis

Algorithms are designed to solve problems, but a given problem can have many different solutions. How then are we to determine which solution is the most efficient for a given problem? One approach is to measure the execution time. We can implement the solution by constructing a computer program, using a given programming language. We then execute the program and time it using a wall clock or the computer's internal clock.

The execution time is dependent on several factors. First, the amount of data that must be processed directly affects the execution time. As the data set size increases, so does the execution time. Second, the execution times can vary depending on the type of hardware and the time of day a computer is used. If we use a multi-process, multi-user system to execute the program, the execution of other programs on the same machine can directly affect the execution time of our program. Finally, the choice of programming language and compiler used to implement an algorithm can also influence the execution time. Some compilers are better optimizers than others and some languages produce better optimized code than others. Thus, we need a method to analyze an algorithm's efficiency independent of the implementation details.

4.1 Complexity Analysis

To determine the efficiency of an algorithm, we can examine the solution itself and measure those aspects of the algorithm that most critically affect its execution time. For example, we can count the number of logical comparisons, data interchanges, or arithmetic operations. Consider the following algorithm for computing the sum of each row of an $n \times n$ matrix and an overall sum of the entire matrix:

```
totalSum = 0                          # Version 1
for i in range( n ) :
  rowSum[i] = 0
  for j in range( n ) :
    rowSum[i] = rowSum[i] + matrix[i,j]
    totalSum = totalSum + matrix[i,j]
```

Suppose we want to analyze the algorithm based on the number of additions performed. In this example, there are only two addition operations, making this a simple task. The algorithm contains two loops, one nested inside the other. The inner loop is executed n times and since it contains the two addition operations, there are a total of $2n$ additions performed by the inner loop for each iteration of the outer loop. The outer loop is also performed n times, for a total of $2n^2$ additions.

Can we improve upon this algorithm to reduce the total number of addition operations performed? Consider a new version of the algorithm in which the second addition is moved out of the inner loop and modified to sum the entries in the `rowSum` array instead of individual elements of the matrix.

```
totalSum = 0                          # Version 2
for i in range( n ) :
  rowSum[i] = 0
  for j in range( n ) :
    rowSum[i] = rowSum[i] + matrix[i,j]
  totalSum = totalSum + rowSum[i]
```

In this version, the inner loop is again executed n times, but this time, it only contains one addition operation. That gives a total of n additions for each iteration of the outer loop, but the outer loop now contains an addition operator of its own. To calculate the total number of additions for this version, we take the n additions performed by the inner loop and add one for the addition performed at the bottom of the outer loop. This gives $n + 1$ additions for each iteration of the outer loop, which is performed n times for a total of $n^2 + n$ additions.

If we compare the two results, it's obvious the number of additions in the second version is less than the first for any n greater than 1. Thus, the second version will execute faster than the first, but the difference in execution times will not be significant. The reason is that both algorithms execute on the same order of magnitude, namely n^2. Thus, as the size of n increases, both algorithms increase at approximately the same rate (though one is slightly better), as illustrated numerically in Table 4.1 and graphically in Figure 4.1.

n	$2n^2$	$n^2 + n$
10	200	110
100	20,000	10,100
1000	2,000,000	1,001,000
10000	200,000,000	100,010,000
100000	20,000,000,000	10,000,100,000

Table 4.1: Growth rate comparisons for different input sizes.

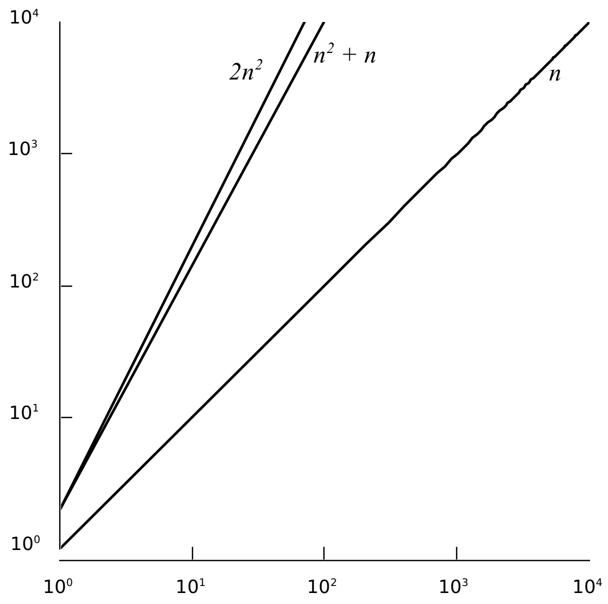


Figure 4.1: Graphical comparison of the growth rates from Table 4.1.

4.1.1 Big-O Notation

Instead of counting the precise number of operations or steps, computer scientists are more interested in classifying an algorithm based on the ***order of magnitude*** as applied to execution time or space requirements. This classification approximates the actual number of required steps for execution or the actual storage requirements in terms of variable-sized data sets. The term ***big-O***, which is derived from the expression "on the order of," is used to specify an algorithm's classification.

Defining Big-O

Assume we have a function $T(n)$ that represents the approximate number of steps required by an algorithm for an input of size n. For the second version of our algorithm in the previous section, this would be written as

$$T_2(n) = n^2 + n$$

Now, suppose there exists a function $f(n)$ defined for the integers $n \geq 0$, such that for some constant c, and some constant m,

$$T(n) \leq cf(n)$$

for all sufficiently large values of $n \geq m$. Then, such an algorithm is said to have a ***time-complexity*** of, or executes on the order of, $f(n)$ relative to the number of operations it requires. In other words, there is a positive integer m and a constant c (***constant of proportionality***) such that for all $n \geq m$, $T(n) \leq cf(n)$. The function $f(n)$ indicates the rate of growth at which the run time of an algorithm increases as the input size, n, increases. To specify the time-complexity of an algorithm, which runs on the order of $f(n)$, we use the notation

$$O(\ f(n)\ )$$

Consider the two versions of our algorithm from earlier. For version one, the time was computed to be $T_1(n) = 2n^2$. If we let $c = 2$, then

$$2n^2 \leq 2n^2$$

for a result of $O(n^2)$. For version two, we computed a time of $T_2(n) = n^2 + n$. Again, if we let $c = 2$, then

$$n^2 + n \leq 2n^2$$

for a result of $O(n^2)$. In this case, the choice of c comes from the observation that when $n \geq 1$, we have $n \leq n^2$ and $n^2 + n \leq n^2 + n^2$, which satisfies the equation in the definition of big-O.

The function $f(n) = n^2$ is not the only choice for satisfying the condition $T(n) \leq cf(n)$. We could have said the algorithms had a run time of $O(n^3)$ or $O(n^4)$ since $2n^2 \leq n^3$ and $2n^2 \leq n^4$ when $n > 1$. The objective, however, is to find a function $f(\cdot)$ that provides the tightest (lowest) ***upper bound*** or limit for the run time of an algorithm. The big-O notation is intended to indicate an algorithm's efficiency for large values of n. There is usually little difference in the execution times of algorithms when n is small.

Constant of Proportionality

The constant of proportionality is only crucial when two algorithms have the same $f(n)$. It usually makes no difference when comparing algorithms whose growth rates are of different magnitudes. Suppose we have two algorithms, L_1 and L_2, with run times equal to n^2 and $2n$ respectively. L_1 has a time-complexity of $O(n^2)$ with $c = 1$ and L_2 has a time of $O(n)$ with $c = 2$. Even though L_1 has a smaller constant of proportionality, L_1 is still slower and, in fact an order of magnitude slower, for large values of n. Thus, $f(n)$ dominates the expression $cf(n)$ and the run time performance of the algorithm. The differences between the run times of these two algorithms is shown numerically in Table 4.2 and graphically in Figure 4.2.

Constructing T(n)

Instead of counting the number of logical comparisons or arithmetic operations, we evaluate an algorithm by considering every operation. For simplicity, we assume that each basic operation or statement, at the abstract level, takes the same amount of time and, thus, each is assumed to cost ***constant time***. The total number of

n	n^2	$2n$
10	100	20
100	10,000	200
1000	1,000,000	2,000
10000	100,000,000	20,000
100000	10,000,000,000	200,000

Table 4.2: Numerical comparison of two sample algorithms.

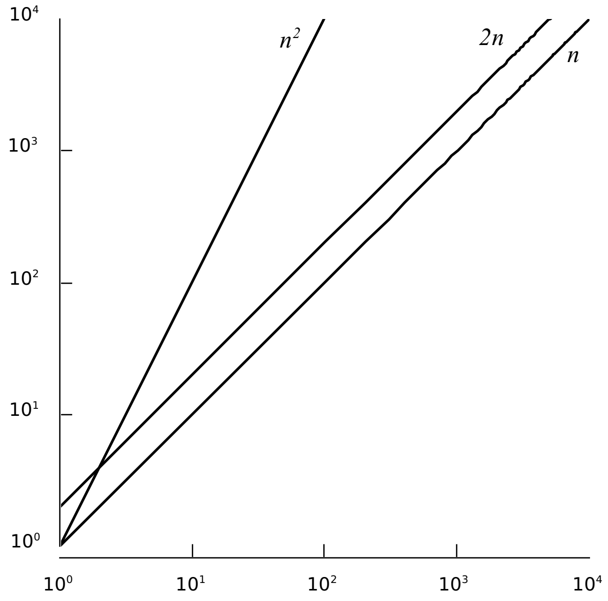
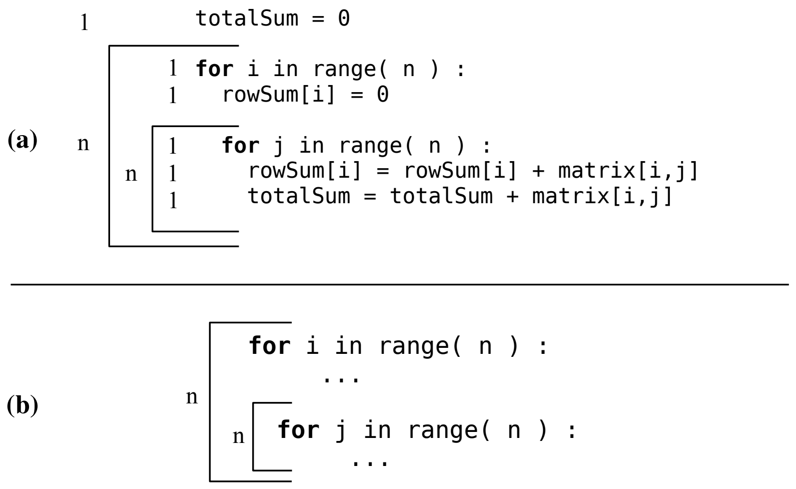


Figure 4.2: Graphical comparison of the data from Table 4.2.

operations required by an algorithm can be computed as a sum of the times required to perform each step:

$$T(n) = f_1(n) + f_2(n) + \ldots + f_k(n).$$

The steps requiring constant time are generally omitted since they eventually become part of the constant of proportionality. Consider Figure 4.3(a), which shows a markup of version one of the algorithm from earlier. The basic operations are marked with a constant time while the loops are marked with the appropriate total number of iterations. Figure 4.3(b) shows the same algorithm but with the constant steps omitted since these operations are independent of the data set size.



(a)

```
1         totalSum = 0
n     1 for i in range( n ) :
      1   rowSum[i] = 0
   n  1   for j in range( n ) :
      1     rowSum[i] = rowSum[i] + matrix[i,j]
      1     totalSum = totalSum + matrix[i,j]
```

(b)

```
n      for i in range( n ) :
          ...
   n    for j in range( n ) :
           ...
```

Figure 4.3: Markup for version one of the matrix summing algorithm: (a) shows all operations marked with the appropriate time and (b) shows only the non-constant time steps.

Choosing the Function

The function $f(n)$ used to categorize a particular algorithm is chosen to be the ***dominant term*** within $T(n)$. That is, the term that is so large for big values of n, that we can ignore the other terms when computing a big-O value. For example, in the expression

$$n^2 + log_2 n + 3n$$

the term n^2 dominates the other terms since for $n \geq 3$, we have

$$\begin{aligned} n^2 + \log_2 n + 3n &\leq n^2 + n^2 + n^2 \\ n^2 + \log_2 n + 3n &\leq 3n^2 \end{aligned}$$

which leads to a time-complexity of $O(n^2)$. Now, consider the function $T(n) = 2n^2 + 15n + 500$ and assume it is the polynomial that represents the exact number of instructions required to execute some algorithm. For small values of n (less than 16), the constant value 500 dominates the function, but what happens as n gets larger, say $100,000$? The term n^2 becomes the dominant term, with the other two becoming less significant in computing the final result.

Classes of Algorithms

We will work with many different algorithms in this text, but most will have a time-complexity selected from among a common set of functions, which are listed in Table 4.3 and illustrated graphically in Figure 4.4.

Algorithms can be classified based on their big-O function. The various classes are commonly named based upon the dominant term. A ***logarithmic*** algorithm is

$f(\cdot)$	**Common Name**
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	log linear
n^2	quadratic
n^3	cubic
a^n	exponential

Table 4.3: Common big-O functions listed from smallest to largest order of magnitude.

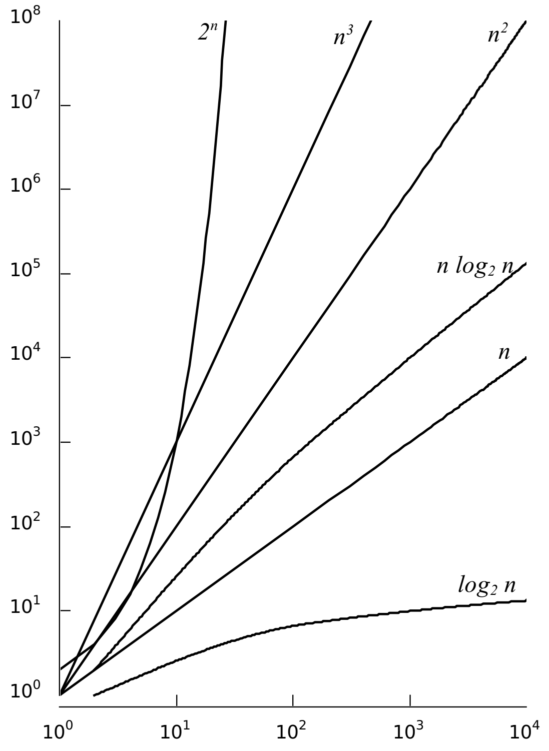


Figure 4.4: Growth rates of the common time-complexity functions.

any algorithm whose time-complexity is $O(\log_a n)$. These algorithms are generally very efficient since $\log_a n$ will increase more slowly than n. For many problems encountered in computer science a will typically equal 2 and thus we use the notation $\log n$ to imply $\log_2 n$. Logarithms of other bases will be explicitly stated. ***Polynomial*** algorithms with an efficiency expressed as a polynomial of the form

$$a_m n^m + a_{m-1} n^{m-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

are characterized by a time-complexity of $O(n^m)$ since the dominant term is the highest power of n. The most common polynomial algorithms are ***linear*** $(m = 1)$, ***quadratic*** $(m = 2)$, and ***cubic*** $(m = 3)$. An algorithm whose efficiency is characterized by a dominant term in the form a^n is called ***exponential***. Exponential algorithms are among the worst algorithms in terms of time-complexity.

4.1.2 Evaluating Python Code

As indicated earlier, when evaluating the time complexity of an algorithm or code segment, we assume that basic operations only require constant time. But what exactly is a basic operation? The ***basic operations*** include statements and function calls whose execution time does not depend on the specific values of the data that is used or manipulated by the given instruction. For example, the assignment statement

```
x = 5
```

is a basic instruction since the time required to assign a reference to the given variable is independent of the value or type of object specified on the righthand side of the `=` sign. The evaluation of arithmetic and logical expressions

```
y = x
z = x + y * 6
done = x > 0 and x < 100
```

are basic instructions, again since they require the same number of steps to perform the given operations regardless of the values of their operands. The subscript operator, when used with Python's sequence types (strings, tuples, and lists) is also a basic instruction.

Linear Time Examples

Now, consider the following assignment statement:

```
y = ex1(n)
```

An assignment statement only requires constant time, but that is the time required to perform the actual assignment and does not include the time required to execute any function calls used on the righthand side of the assignment statement.

To determine the run time of the previous statement, we must know the cost of the function call `ex1(n)`. The time required by a function call is the time it takes to execute the given function. For example, consider the `ex1()` function, which computes the sum of the integer values in the range $[0 \ldots n)$:

```
def ex1( n ):
  total = 0
  for i in range( n ) :
    total += i
  return total
```

NOTE

Efficiency of String Operations. Most of the string operations have a time-complexity that is proportional to the length of the string. For most problems that do not involve string processing, string operations seldom have an impact on the run time of an algorithm. Thus, in the text, we assume the string operations, including the use of the `print()` function, only require constant time, unless explicitly stated otherwise.

The time required to execute a loop depends on the number of iterations performed and the time needed to execute the loop body during each iteration. In this case, the loop will be executed n times and the loop body only requires constant time since it contains a single basic instruction. (Note that the underlying mechanism of the `for` loop and the `range()` function are both $O(1)$.) We can compute the time required by the loop as $T(n) = n * 1$ for a result of $O(n)$.

But what about the other statements in the function? The first line of the function and the `return` statement only require constant time. Remember, it's common to omit the steps that only require constant time and instead focus on the critical operations, those that contribute to the overall time. In most instances, this means we can limit our evaluation to repetition and selection statements and function and method calls since those have the greatest impact on the overall time of an algorithm. Since the loop is the only non-constant step, the function `ex1()` has a run time of $O(n)$. That means the statement `y = ex1(n)` from earlier requires linear time. Next, consider the following function, which includes two `for` loops:

```
def ex2( n ):
  count = 0
  for i in range( n ) :
    count += 1
  for j in range( n ) :
    count += 1
  return count
```

To evaluate the function, we have to determine the time required by each loop. The two loops each require $O(n)$ time as they are just like the loop in function `ex1()` earlier. If we combine the times, it yields $T(n) = n + n$ for a result of $O(n)$.

Quadratic Time Examples

When presented with nested loops, such as in the following, the time required by the inner loop impacts the time of the outer loop.

```
def ex3( n ):
  count = 0
  for i in range( n ) :
    for j in range( n ) :
      count += 1
  return count
```

Both loops will be executed n, but since the inner loop is nested inside the outer loop, the total time required by the outer loop will be $T(n) = n * n$, resulting in a time of $O(n^2)$ for the `ex3()` function. Not all nested loops result in a quadratic time. Consider the following function:

```
def ex4( n ):
  count = 0
  for i in range( n ) :
    for j in range( 25 ) :
      count += 1
  return count
```

which has a time-complexity of $O(n)$. The function contains a nested loop, but the inner loop executes independent of the size variable n. Since the inner loop executes a constant number of times, it is a constant time operation. The outer loop executes n times, resulting in a linear run time. The next example presents a special case of nested loops:

```
def ex5( n ):
  count = 0
  for i in range( n ) :
    for j in range( i+1 ) :
      count += 1
  return count
```

How many times does the inner loop execute? It depends on the current iteration of the outer loop. On the first iteration of the outer loop, the inner loop will execute one time; on the second iteration, it executes two times; on the third iteration, it executes three times, and so on until the last iteration when the inner loop will execute n times. The time required to execute the outer loop will be the number of times the increment statement `count += 1` is executed. Since the inner loop varies from 1 to n iterations by increments of 1, the total number of times the increment statement will be executed is equal to the sum of the first n positive integers:

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

which results in a quadratic time of $O(n^2)$.

Logarithmic Time Examples

The next example contains a single loop, but notice the change to the modification step. Instead of incrementing (or decrementing) by one, it cuts the loop variable in half each time through the loop.

```
def ex6( n ):
  count = 0
  i = n
  while i >= 1 :
    count += 1
    i = i // 2
  return count
```

To determine the run time of this function, we have to determine the number of loop iterations just like we did with the earlier examples. Since the loop variable is cut in half each time, this will be less than n. For example, if n equals 16, variable `i` will contain the following five values during subsequent iterations (16, 8, 4, 2, 1).

Given a small number, it's easy to determine the number of loop iterations. But how do we compute the number of iterations for any given value of n? When the size of the input is reduced by half in each subsequent iteration, the number of iterations required to reach a size of one will be equal to

$$\lfloor \log_2 n \rfloor + 1$$

or the largest integer less than $\log_2 n$, plus 1. In our example of $n = 16$, there are $\log_2 16 + 1$, or four iterations. The logarithm to base a of a number n, which is normally written as $y = log_a n$, is the power to which a must be raised to equal n, $n = a^y$. Thus, function `ex6()` requires $O(\log n)$ time. Since many problems in computer science that repeatedly reduce the input size do so by half, it's not uncommon to use $\log n$ to imply $\log_2 n$ when specifying the run time of an algorithm.

Finally, consider the following definition of function `ex7()`, which calls `ex6()` from within a loop. Since the loop is executed n times and function `ex6()` requires logarithmic time, `ex7()` will have a run time of $O(n \log n)$.

```
def ex7( n ):
  count = 0
  for i in range( n )
    count += ex6( n )
  return count
```

Different Cases

Some algorithms can have run times that are different orders of magnitude for different sets of inputs of the same size. These algorithms can be evaluated for their best, worst, and average cases. Algorithms that have different cases can typically be identified by the inclusion of an event-controlled loop or a conditional statement. Consider the following example, which traverses a list containing integer values to find the position of the first negative value. Note that for this problem, the input is the collection of n values contained in the list.

```
def findNeg( intList ):
  n = len(intList)
  for i in range( n ) :
    if intList[i] < 0 :
      return i
  return None
```

At first glance, it appears the loop will execute n times, where n is the size of the list. But notice the `return` statement inside the loop, which can cause it to terminate early. If the list does not contain a negative value,

```
L = [ 72, 4, 90, 56, 12, 67, 43, 17, 2, 86, 33 ]
p = findNeg( L )
```

the `return` statement inside the loop will not be executed and the loop will terminate in the normal fashion from having traversed all n times. In this case, the function requires $O(n)$ time. This is known as the ***worst case*** since the function must examine every value in the list requiring the most number of steps. Now consider the case where the list contains a negative value in the first element:

```
L = [ -12, 50, 4, 67, 39, 22, 43, 2, 17, 28 ]
p = findNeg( L )
```

There will only be one iteration of the loop since the test of the condition by the `if` statement will be true the first time through and the `return` statement inside the loop will be executed. In this case, the `findNeg()` function only requires $O(1)$ time. This is known as the ***best case*** since the function only has to examine the first value in the list requiring the least number of steps.

The ***average case*** is evaluated for an expected data set or how we expect the algorithm to perform on average. For the `findNeg()` function, we would expect the search to iterate halfway through the list before finding the first negative value, which on average requires $n/2$ iterations. The average case is more difficult to evaluate because it's not always readily apparent what constitutes the average case for a particular problem.

In general, we are more interested in the worst case time-complexity of an algorithm as it provides an upper bound over all possible inputs. In addition, we can compare the worst case run times of different implementations of an algorithm to determine which is the most efficient for any input.

4.2 Evaluating the Python List

We defined several abstract data types for storing and using collections of data in the previous chapters. The next logical step is to analyze the operations of the various ADTs to determine their efficiency. The result of this analysis depends on the efficiency of the Python list since it was the primary data structure used to implement many of the earlier abstract data types.

The implementation details of the list were discussed in Chapter 2. In this section, we use those details and evaluate the efficiency of some of the more common operations. A summary of the worst case run times are shown in Table 4.4.

List Operation	Worst Case
`v = list()`	$O(1)$
`v = [ 0 ] * n`	$O(n)$
`v[i] = x`	$O(1)$
`v.append(x)`	$O(n)$
`v.extend(w)`	$O(n)$
`v.insert(x)`	$O(n)$
`v.pop()`	$O(n)$
traversal	$O(n)$

Table 4.4: Worst case time-complexities for the more common list operations.

List Traversal

A sequence traversal accesses the individual items, one after the other, in order to perform some operation on every item. Python provides the built-in iteration for the list structure, which accesses the items in sequential order starting with the first item. Consider the following code segment, which iterates over and computes the sum of the integer values in a list:

```
sum = 0
for value in valueList :
  sum = sum + value
```

To determine the order of complexity for this simple algorithm, we must first look at the internal implementation of the traversal. Iteration over the contiguous elements of a 1-D array, which is used to store the elements of a list, requires a count-controlled loop with an index variable whose value ranges over the indices of the subarray. The list iteration above is equivalent to the following:

```
sum = 0
for i in range( len(valueList) ) :
  sum = sum + valueList[i]
```

Assuming the sequence contains n items, it's obvious the loop performs n iterations. Since all of the operations within the loop only require constant time, including the element access operation, a complete list traversal requires $O(n)$ time. Note, this time establishes a minimum required for a complete list traversal. It can actually be higher if any operations performed during each iteration are worse than constant time, unlike this example.

List Allocation

Creating a list, like the creation of any object, is considered an operation whose time-complexity can be analyzed. There are two techniques commonly used to create a list:

```
temp = list()
valueList = [ 0 ] * n
```

The first example creates an empty list, which can be accomplished in constant time. The second creates a list containing n elements, with each element initialized to 0. The actual allocation of the n elements can be done in constant time, but the initialization of the individual elements requires a list traversal. Since there are n elements and a traversal requires linear time, the allocation of a vector with n elements requires $O(n)$ time.

Appending to a List

The `append()` operation adds a new item to the end of the sequence. If the underlying array used to implement the list has available capacity to add the new item, the operation has a best case time of $O(1)$ since it only requires a single element access. In the worst case, there are no available slots and the array has to be expanded using the steps described in Section 2.2. Creating the new larger array and destroying the old array can each be done in $O(1)$ time. To copy the contents of the old array to the new larger array, the items have to be copied element by element, which requires $O(n)$ time. Combining the times from the three steps yields a time of $T(n) = 1 + 1 + n$ and a worst case time of $O(n)$.

Extending a List

The `extend()` operation adds the entire contents of a source list to the end of the destination list. This operation involves two lists, each of which have their own collection of items that may be of different lengths. To simplify the analysis, however, we can assume both lists contain n items. When the destination list has sufficient capacity to store the new items, the entire contents of the source list can be copied in $O(n)$ time. But if there is not sufficient capacity, the underlying array of the destination list has to be expanded to make room for the new items. This expansion requires $O(n)$ time since there are currently n items in the destination list. After the expansion, the n items in the source list are copied to the expanded array, which also requires $O(n)$ time. Thus, in the worst case the extend operation requires $T(n) = n + n = 2n$ or $O(n)$ time.

Inserting and Removing Items

Inserting a new item into a list is very similar to appending an item except the new item can be placed anywhere within the list, possibly requiring a shift in elements. An item can be removed from any element within a list, which may also involve shifting elements. Both of these operations require linear time in the worst case, the proof of which is left as an exercise.

4.3 Amortized Cost

The `append()` operation of the list structure introduces a special case in algorithm analysis. The time required depends on the available capacity of the underlying array used to implement the list. If there are available slots, a value can be appended to the list in constant time. If the array has to be expanded to make room for the new value, however, the append operation takes linear time. When the array is expanded, extra capacity is added that can be used to add more items without having to immediately expand the array. Thus, the number of times the `append()` operation actually requires linear time in a sequence of n operations depends on the strategy used to expand the underlying array. Consider the problem in which a sequence of n append operations are performed on an initially empty list, where n is a power of 2.

```
L = list()
for i in range( 1, n+1 ) :
  L.append( i )
```

Suppose the array is doubled in capacity each time it has to be expanded and assume the size of the underlying array for an empty list has the capacity for a single item. We can tally or compute the total running time for this problem by considering the time required for each individual append operation. This approach is known as the ***aggregate method*** since it computes the total from the individual operations.

Table 4.5 illustrates the aggregate method when applied to a sequence of 16 append operations. s_i represents the time required to physically store the i^{th} value when there is an available slot in the array or immediately after the array has been expanded. Storing an item into an array element is a constant time operation. e_i represents the time required to expand the array when it does not contain available capacity to store the item. Based on our assumptions related to the size of the array, an expansion only occurs when $i - 1$ is a power of 2 and the time incurred is based on the current size of the array $(i - 1)$. While every append operation entails a storage cost, relatively few require an expansion cost. Note that as the size of n increases, the distance between append operations requiring an expansion also increases.

Based on the tabulated results in Table 4.5, the total time required to perform a sequence of 16 append operations on an initially empty list is 31, or just under $2n$. This results from a total storage cost (s_i) of 16 and a total expansion cost (e_i) of 15. It can be shown that for any n, the sum of the storage and expansion costs, $s_i + e_i$, will never be more than $T(n) = 2n$. Since there are relatively few expansion operations, the expansion cost can be distributed across the sequence of operations, resulting in an ***amortized cost*** of $T(n) = 2n/n$ or $O(1)$ for the append operation.

Amortized analysis is the process of computing the time-complexity for a sequence of operations by computing the average cost over the entire sequence. For this technique to be applied, the cost per operation must be known and it must

i	s_i	e_i	**Size**	**List Contents**
1	1	-	1	1
2	1	1	2	1 2
3	1	2	4	1 2 3 ☐
4	1	-	4	1 2 3 4
5	1	4	8	1 2 3 4 5 ☐ ☐ ☐
6	1	-	8	1 2 3 4 5 6 ☐ ☐
7	1	-	8	1 2 3 4 5 6 7 ☐
8	1	-	8	1 2 3 4 5 6 7 8
9	1	8	16	1 2 3 4 5 6 7 8 9 ☐ ☐ ☐ ☐ ☐ ☐ ☐
10	1	-	16	1 2 3 4 5 6 7 8 9 10 ☐ ☐ ☐ ☐ ☐ ☐
11	1	-	16	1 2 3 4 5 6 7 8 9 10 11 ☐ ☐ ☐ ☐ ☐
12	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 ☐ ☐ ☐ ☐
13	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 ☐ ☐ ☐
14	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 ☐ ☐
15	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ☐
16	1	-	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Table 4.5: Using the aggregate method to compute the total run time for a sequence of 16 append operations.

vary in which many of the operations in the sequence contribute little cost and only a few operations contribute a high cost to the overall time. This is exactly the case with the `append()` method. In a long sequence of append operations, only a few instances require $O(n)$, while many of them are $O(1)$. The amortized cost can only be used for a long sequence of append operations. If an algorithm used a single append operation, the cost for that one operation is still $O(n)$ in the worst case since we do not know if that's the instance that causes the underlying array to be expanded.

CAUTION

⚠ **Amortized Cost Is Not Average Case Time.** Do not confuse amortized cost with that of average case time. In average case analysis, the evaluation is done by computing an average over all possible inputs and sometimes requires the use of statistics. Amortized analysis computes an average cost over a sequence of operations in which many of those operations are "cheap" and relatively few are "expensive" in terms of contributing to the overall time.

4.4 Evaluating the Set ADT

We can use complexity analysis to determine the efficiency of the Set ADT operations as implemented in Section 3.1. For convenience, the relevant portions of that implementation are shown again in Listing 4.1 on the next page. The evaluation is quite simple since the ADT was implemented using the list and we just evaluated the methods for that structure. Table 4.6 provides a summary of the worst case time-complexities for those operations implemented earlier in the text.

Operation	Worst Case
`s = Set()`	$O(1)$
`len(s)`	$O(1)$
`x in s`	$O(n)$
`s.add(x)`	$O(n)$
`s.isSubsetOf(t)`	$O(n^2)$
`s == t`	$O(n^2)$
`s.union(t)`	$O(n^2)$
traversal	$O(n)$

Table 4.6: Time-complexities for the Set ADT implementation using an unsorted list.

Simple Operations

Evaluating the constructor and length operation is straightforward as they simply call the corresponding list operation. The `__contains__` method, which determines if an element is contained in the set, uses the `in` operator to perform a linear search over the elements stored in the underlying list. The search operation, which requires $O(n)$ time, will be presented in the next section and we postpone its analysis until that time. The `add()` method also requires $O(n)$ time in the worst case since it uses the `in` operator to determine if the element is unique and the `append()` method to add the unique item to the underlying list, both of which require linear time in the worst case.

Operations of Two Sets

The remaining methods of the `Set` class involve the use of two sets, which we label A and B, where A is the `self` set and B is the argument passed to the given method. To simplify the analysis, we assume each set contains n elements. A more complete analysis would involve the use of two variables, one for the size of each set. But the analysis of this more specific case is sufficient for our purposes.

The `isSubsetOf()` method determines if A is a subset of B. It iterates over the n elements of set A, during which the `in` operator is used to determine if the

Listing 4.1 A partial listing of the `linearset.py` module from Listing 3.1.

```python
class Set :
  def __init__( self ):
    self._theElements = list()

  def __len__( self ):
    return len( self._theElements )

  def __contains__( self, element ):
    return element in self._theElements

  def add( self, element ):
    if element not in self :
      self._theElements.append( element )

  def remove( self, element ):
    assert element in self, "The element must be in the set."
    self._theElements.remove( item )

  def __eq__( self, setB ):
    if len( self ) != len( setB ) :
      return False
    else :
      return self.isSubsetOf( setB )

  def isSubsetOf( self, setB ):
    for element in self :
      if element not in setB :
        return False
    return True

  def union( self, setB ):
    newSet = Set()
    newSet._theElements.extend( self._theElements )
    for element in setB :
      if element not in self :
        newSet._theElements.append( element )
    return newSet
```

given element is a member of set B. Since there are n repetitions of the loop and each use of the `in` operator requires $O(n)$ time, the `isSubsetOf()` method has a quadratic run time of $O(n^2)$. The set equality operation is also $O(n^2)$ since it calls `isSubsetOf()` after determining the two sets are of equal size.

Set Union Operation

The set `union()` operation creates a new set, C, that contains all of the unique elements from both sets A and B. It requires three steps. The first step creates the new set C, which can be done in constant time. The second step fills set C with the elements from set A, which requires $O(n)$ time since the `extend()` list method is used to add the elements to C. The last step iterates over the elements of set B during which the `in` operator is used to determine if the given element is a member of set A. If the element is not a member of set A, it's added to set C by applying the `append()` list method. We know from earlier the linear search performed by the `in` operator requires $O(n)$ time and we can use the $O(1)$ amortized cost of the `append()` method since it is applied in sequence. Given that the loop is performed n times and each iteration requires $n + 1$ time, this step requires $O(n^2)$ time. Combining the times for the three steps yields a worst case time of $O(n^2)$.

4.5 Application: The Sparse Matrix

A matrix containing a large number of zero elements is called a ***sparse matrix***. Sparse matrices are very common in scientific applications, especially those dealing with systems of linear equations. A sparse matrix is formally defined to be an $m \times n$ matrix that contains k non-zero elements such that $k \ll m \times n$. The 2-D array data structure used to implement the Matrix ADT in Chapter 2 works well for general matrices. But when used to store huge sparse matrices, large amounts of memory can be wasted and the operations can be inefficient since the zero elements are also stored in the 2-D array.

Consider the sample 5×8 sparse matrix in Figure 4.5. Is there a different structure or organization we can use to store the elements of a sparse matrix that does not waste space? One approach is to organize and store the non-zero elements of the matrix within a single list instead of a 2-D array.

$$\begin{bmatrix} \cdot & 3 & \cdot & \cdot & 8 & \cdot & \cdot & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & \cdot & 5 & \cdot \\ \cdot & \cdot & 9 & \cdot & \cdot & 2 & \cdot & \cdot \\ \cdot & 7 & \cdot & \cdot & \cdot & \cdot & \cdot & 3 \\ \cdot & \cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot \end{bmatrix}$$

Figure 4.5: A sample sparse matrix with zero elements indicated with dots.

4.5.1 List-Based Implementation

In this section, we define and implement a class for storing and working with sparse matrices in which the non-zero elements are stored in a list. The operations of a sparse matrix are the same as those for a general matrix and many of them can be implemented in a similar fashion as was done for the `Matrix` class in Listing 2.3. This would be sufficient if our only objective was to reduce the storage cost, but we can take advantage of only storing the non-zero elements to improve the efficiency of several of the sparse matrix operations. The implementation of the `SparseMatrix` class is provided in Listing 4.2 on the next page. Note the use of the new class name to distinguish this version from the original Matrix ADT and to indicate it is meant for use with sparse matrices. A sample instance of the class that corresponds to the sparse matrix from Figure 4.5 is illustrated in Figure 4.6.

Listing 4.2 The `sparsematrix.py` module.

```
# Implementation of the Sparse Matrix ADT using a list.

class SparseMatrix :
   # Create a sparse matrix of size numRows x numCols initialized to 0.
  def __init__( self, numRows, numCols ):
    self._numRows = numRows
    self._numCols = numCols
    self._elementList = list()

   # Return the number of rows in the matrix.
  def numRows( self ):
    return self._numRows

   # Return the number of columns in the matrix.
  def numCols( self ):
    return self._numCols

   # Return the value of element (i, j): x[i,j]
  def __getitem__( self, ndxTuple ):
    ......

   # Set the value of element (i,j) to the value s: x[i,j] = s
  def __setitem__( self, ndxTuple, scalar ):
    ndx = self._findPosition( ndxTuple[0], ndxTuple[1] )
    if ndx is not None :  # if the element is found in the list.
      if scalar != 0.0 :
        self._elementList[ndx].value = scalar
      else :
        self._elementList.pop( ndx )
    else :          # if the element is zero and not in the list.
      if scalar != 0.0 :
        element = _MatrixElement( ndxTuple[0], ndxTuple[1], scalar )
        self._elementList.append( element )

   # Scale the matrix by the given scalar.
  def scaleBy( self, scalar ):
    for element in self._elementList :
      element.value *= scalar

  # The additional methods should be placed here.....
  # def __add__( self, rhsMatrix ):
  # def __sub__( self, rhsMatrix ):
  # def __mul__( self, rhsMatrix ):

   # Helper method used to find a specific matrix element (row,col) in the
   # list of non-zero entries. None is returned if the element is not found.
  def _findPosition( self, row, col ):
    n = len( self._elementList )
    for i in range( n ) :
      if row == self._elementList[i].row and \
         col == self._elementList[i].col:
        return i         # return the index of the element if found.
    return None       # return None when the element is zero.
```

```
# Storage class for holding the non-zero matrix elements.
class _MatrixElement:
  def __init__( self, row, col, value ):
    self.row = row
    self.col = col
    self.value = value
```

Constructor

The constructor defines three attributes for storing the data related to the sparse matrix. The `_elementList` field stores `_MatrixElement` objects representing the non-zero elements. Instances of the storage class contain not only the value for a specific element but also the row and column indices indicating its location within the matrix. The `_numRows` and `_numCols` fields are used to store the dimensions of the matrix. This information cannot be obtained from the element list as was done with the `Array2D` used in the implementation of the Matrix ADT in Chapter 2.

Helper Method

Since the element list only contains the non-zero entries, accessing an individual element is no longer as simple as directly referencing an element of the rectangular grid. Instead, we must search through the list to locate a specific non-zero element. The helper method `_findPosition()` performs this linear search by iterating through the element list looking for an entry with the given row and column indices. If found, it returns the list index of the cell containing the element; otherwise, `None` is returned to indicate the absence of the element.

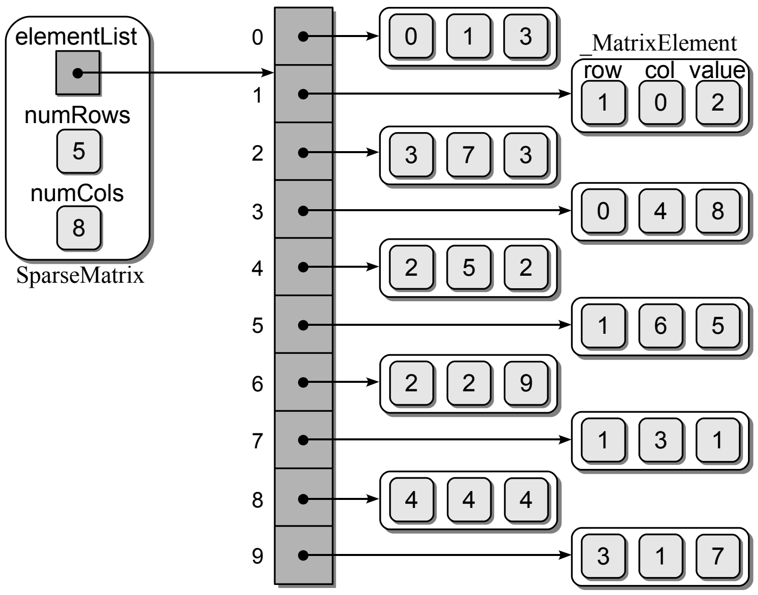


Figure 4.6: A list of `_MatrixElement` objects representing a sparse matrix.

Modifying an Element

The `__setitem__` method for the `SparseMatrix` class is a bit more involved than that for the `Matrix` class. The value of an element cannot be directly set as was done when using the 2-D array. Instead, there are four possible conditions:

1. The element is in the list (and thus non-zero) and the new value is non-zero.
2. The element is in the list, but the new value is zero, turning the element into a zero element.
3. The element is not currently in the list and the new value is non-zero.
4. The element is not currently in the list, and the new value is zero.

The step in implementing the `__setitem__` method, as shown in lines 23–33 of Listing 4.2, is to determine if the element is in the list using the `_findPosition()` helper method. If the entry is in the list, we either change the corresponding element to the new value if it is non-zero or we remove the entry from the list when the new value is zero. On the other hand, if there is no entry for the given element, then a new `_MatrixElement` object must be created and appended to the list. Of course, this is only done if the new value is non-zero.

Matrix Scaling

Scaling a matrix requires multiplying each element of the matrix by a given scale factor. Since the zero elements of the matrix are not affected by the scale factor, the implementation of this operation for the sparse matrix is as simple as traversing the list of `_MatrixElement` objects and scaling the corresponding value.

Matrix Addition

In the `add()` method of the `Matrix` class implemented in Chapter 2, we iterated over the 2-D array and added the values, element by element, and stored the results in the corresponding element of the new matrix. We could use the same loop structure shown here for the `SparseMatrix` class:

```
 # Add the corresponding elements in the two matrices.
for r in range( self.numRows() ) :
  for c in range( self.numCols() ) :
    newMatrix[ r, c ] = self[ r, c ] + rhsMatrix[ r, c ]
return newMatrix
```

Given a matrix of size $n \times n$, this implementation of the add operation requires $O(n^2)$ time. If the sparse matrix contains a significant number of zero elements, this can be inefficient. Instead, only the non-zero elements contained in the two sparse matrices must be considered when adding to matrices. The nested loops can be replaced with two separate loops to reduce the number of required iterations. The new solution for sparse matrix addition requires four steps:

1. Verify the size of the two matrices to ensure they are the same as required by matrix addition.
2. Create a new `SparseMatrix` object with the same number of rows and columns as the other two.
3. Duplicate the elements of the `self` matrix and store them in the new matrix.
4. Iterate over the element list of the righthand side matrix (`rhsMatrix`) to add the non-zero values to the corresponding elements in the new matrix.

The implementation of the add operation is provided in Listing 4.3. The first two steps of the add operation are straightforward. The third step of copying the elements of the `self` matrix to the new matrix requires a list duplication, which is handled by the first loop. The second loop handles the fourth step outlined above by iterating over the list of `_MatrixElement` objects in the `rhsMatrix` and adding their values to the corresponding values in the new sparse matrix. Note the use of the `__getitem__` and `__setitem__` methods in the second loop. This is necessary since the two methods properly manage any zero elements that may currently exist in the `newMatrix` or that may result after adding corresponding elements.

Listing 4.3 Implementation of the `SparseMatrix` add operation.

```python
class SparseMatrix :
# ...
  def __add__( self, rhsMatrix ):
    assert rhsMatrix.numRows() == self.numRows() and \
           rhsMatrix.numCols() == self.numCols(), \
      "Matrix sizes not compatible for the add operation."

     # Create the new matrix.
    newMatrix = SparseMatrix( self.numRows(), self.numCols() )

     # Duplicate the lhs matrix. The elements are mutable, thus we must
     # create new objects and not simply copy the references.
    for element in self._elementList :
      dupElement = _MatrixElement(element.row, element.col, element.value)
      newMatrix._elementList.append( dupElement )

     # Iterate through each non-zero element of the rhsMatrix.
    for element in rhsMatrix._elementList :
       # Get the value of the corresponding element in the new matrix.
      value = newMatrix[ element.row, element.col ]
      value += element.value
       # Store the new value back to the new matrix.
      newMatrix[ element.row, element.col ] = value

     # Return the new matrix.
    return newMatrix
```

4.5.2 Efficiency Analysis

To evaluate the various operations of the sparse matrix, we can assume a square $n \times n$ matrix since this would be the worst possible case. We begin with the `_findPosition()` helper method, which performs a sequential search over the list of non-zero entries. The worst case occurs when every item in the list is examined. But how many iterations does that require? It depends on the size of the element list. From the definition of a sparse matrix, we know it contains k non-zero elements such that $k \ll n^2$. Thus, the worst case run time of the helper method is $O(k)$.

The `__setitem__` method calls `_findPosition()`, which requires k time. It then changes the value of the target entry, which is a constant time operation, or either removes an entry from the list or appends a new entry. The list operations require k time in the worst case, resulting in an overall time of $O(k)$ for the set operation. The `__getitem__` method can be evaluated in the same fashion and also has a worst case time of $O(k)$.

To evaluate the operations that manipulate two `SparseMatrix` objects, we can specify that both matrices are of the same size or that k will represent the size of the larger of the two lists. Computing the worst case time for the new `add()` method requires that we first determine the complexity of the individual steps.

- The size verification and new matrix creation are constant steps.
- To duplicate the entries of the lefthand side sparse matrix requires k time since `append()` has an amortized cost of $O(1)$.
- The second loop iterates over the element list of the righthand side matrix, which we have assumed also contains k elements. Since the get and set element operations used within the loop each require k time in the worst case, the loop requires $2k * k$ or $2k^2$ time.

Combining this with the time for the previous steps, the add operation is $O(k^2)$ in the worst case. Is this time better than that for the add operation from the `Matrix` class implemented as a 2-D array? That depends on the size of k. If there were no zero elements in either matrix, then $k = n^2$, which results in a worst case time of $O(n^4)$. Remember, however, this implementation is meant to be used with a sparse matrix in which $k \ll m \times n$. In addition, the add operation only depends on the size of the element list, k. Increasing the value of m or n does not increase the size of k. For the analysis of this algorithm, m and n simply provide a maximum value for k and are not variables in the equation.

The use of a list as the underlying data structure to store the non-zero elements of a sparse matrix is a much better implementation than the use of a 2-D array as it can save significant storage space for large matrices. On the other hand, it introduces element access operations that are more inefficient than when using the 2-D array. Table 4.7 provides a comparison of the worst case time-complexities for several of the operations of the `Matrix` class using a 2-D array and the `SparseMatrix` class using a list. In later chapters, we will further explore the Sparse Matrix ADT and attempt to improve the time-complexities of the various operations.

Operation	Matrix	Sparse Matrix
constructor	$O(n^2)$	$O(1)$
`s.numRows()`	$O(1)$	$O(1)$
`s.numCols()`	$O(1)$	$O(1)$
`s.scaleBy(x)`	$O(n^2)$	$O(k)$
`x = s[i,j]`	$O(1)$	$O(k)$
`s[i,j] = x`	$O(1)$	$O(k)$
`r = s + t`	$O(n^2)$	$O(k^2)$

Table 4.7: Comparison of the worst case time-complexities for the `Matrix` class implemented using a 2-D array and the `SparseMatrix` class using a list.

Exercises

4.1 Arrange the following expressions from slowest to fastest growth rate.

$$n \log_2 n \qquad 4^n \qquad k \log_2 n \qquad 5n^2 \qquad 40 \log_2 n \qquad \log_4 n \qquad 12n^6$$

4.2 Determine the $O(\cdot)$ for each of the following functions, which represent the number of steps required for some algorithm.

(a) $T(n) = n^2 + 400n + 5$

(b) $T(n) = 67n + 3n$

(c) $T(n) = 2n + 5n \log n + 100$

(d) $T(n) = \log n + 2n^2 + 55$

(e) $T(n) = 3(2^n) + n^8 + 1024$

(f) $T(n,k) = kn + \log k$

(g) $T(n,k) = 9n + k \log n + 1000$

4.3 What is the time-complexity of the `printCalendar()` function implemented in Exercise 1.3?

4.4 Determine the $O(\cdot)$ for the following `Set` operations implemented in Chapter 1: `difference()`, `intersect()`, and `remove()`.

4.5 What is the time-complexity of the proper subset test operation implemented in Exercise 3.3?

4.6 Prove or show why the worst case time-complexity for the `insert()` and `remove()` list operations is $O(n)$.

4.7 Evaluate each of the following code segments and determine the $O(\cdot)$ for the best and worst cases. Assume an input size of n.

(a)
```
sum = 0
for i in range( n ) :
  if i % 2 == 0 :
    sum += i
```

(b)
```
sum = 0
i = n
while i > 0 :
  sum += i
  i = i / 2
```

(c)
```
for i in range( n ) :
  if i % 3 == 0 :
    for j in range( n / 2 ) :
      sum += j
  elif i % 2 == 0 :
    for j in range( 5 ) :
      sum += j
  else :
    for j in range( n ) :
      sum += j
```

4.8 The slice operation is used to create a new list that contains a subset of items from a source list. Implement the `slice()` function:

```
def slice( theList, first, last )
```

which accepts a list and creates a sublist of the values in `theList`. What is the worst case time for your implementation and what is the best case time?

4.9 Implement the remaining methods of the `SparseMatrix` class: `transpose()`, `__getitem__`, `subtract()`, and `multiply()`.

4.10 Determine the worst case time-complexities for the `SparseMatrix` methods implemented in the previous question.

4.11 Determine the worst case time-complexities for the methods of your `ReversiGameLogic` class implemented in Programming Project 2.4.

4.12 Add Python operator methods to the `SparseMatrix` class that can be used in place of the named methods for several of the operations.

Operator Method	**Current Method**
`__add__(rhsMatrix)`	`add(rhsMatrix)`
`__mul__(rhsMatrix)`	`subtract(rhsMatrix)`
`__sub__(rhsMatrix)`	`multiply(rhsMatrix)`

Programming Projects

4.1 The game of Life is defined for an infinite-sized grid. In Chapter 2, we defined the Life Grid ADT to use a fixed-size grid in which the user specified the width and height of the grid. This was sufficient as an illustration of the use of a 2-D array for the implementation of the game of Life. But a full implementation should allow for an infinite-sized grid. Implement the Sparse Life Grid ADT using an approach similar to the one used to implement the sparse matrix.

- `SparseLifeGrid()`: Creates a new infinite-sized game grid. All cells in the grid are initially set to dead.
- `minRange()`: Returns a 2-tuple (`minrow`, `mincol`) that contains the minimum row index and the minimum column index that is currently occupied by a live cell.
- `maxRange()`: Returns a 2-tuple (`maxrow`, `maxcol`) that contains the maximum row index and the maximum column index that is currently occupied by a live cell.
- `configure( coordList )`: Configures the grid for evolving the first generation. The `coordList` argument is a sequence of 2-tuples with each tuple representing the coordinates (r, c) of the cells to be set as alive. All remaining cells are cleared or set to dead.
- `clearCell( row, col )`: Clears the individual cell (`row`, `col`) and sets it to dead. The cell indices must be within the valid range of the grid.
- `setCell( row, col )`: Sets the indicated cell (`row`, `col`) to be alive. The cell indices must be within the valid range of the grid.
- `isLiveCell( row,col )`: Returns a boolean value indicating if the given cell (`row`, `col`) contains a live organism. The cell indices must be within the valid range of the grid.
- `numLiveNeighbors( row, col )`: Returns the number of live neighbors for the given cell (`row`, `col`). The neighbors of a cell include all of the cells immediately surrounding it in all directions. For the cells along the border of the grid, the neighbors that fall outside the grid are assumed to be dead. The cell indices must be within the valid range of the grid.

4.2 Implement a new version of the `gameoflife.py` program to use your `SparseLifeGrid` class from the previous question.

4.3 Repeat Exercise 2.5 from Chapter 2 but use your new version of the `gameoflife.py` program from the previous question.

4.4 The digital grayscale image was introduced in Programming Project 2.3 and an abstract data type was defined and implemented for storing grayscale images. A color digital image is also a two-dimensional raster image, but unlike the grayscale image, the pixels of a color image store data representing colors instead of a single grayscale value. There are different ways to specify color, but one of the most common is with the use of the discrete RGB color space. Individual colors are specified by three intensity values or components within the range $[0 \ldots 255]$, one for each of the three primary colors that represent the amount of red, green, and blue light that must be added to produce the given color. We can define the `RGBColor` class for use in storing a single color in the discrete RGB color space.

```
class RGBColor :
  def __init__( self, red = 0, green = 0, blue = 0 ):
    self.red = red
    self.green = green
    self.blue = blue
```

Given the description of the operations for the Color Image ADT, implement the abstract data type using a 2-D array that stores instances of the `RGBColor` class. Note when setting the initial color in the constructor or when clearing the image to a specific color, you can store aliases to one `RGBColor` object in each element of the array.

- `ColorImage( nrows, ncols )`: Creates a new instance that consists of `nrows` and `ncols` of pixels each set to black.
- `width()`: Returns the width of the image.
- `height()`: Returns the height of the image.
- `clear( color )`: Clears the entire image by setting each pixel to the given RGB `color`.
- *`getitem`*`( row, col )`: Returns the RGB color of the given pixel as an `RGBColor` object. The pixel coordinates must be within the valid range.
- *`setitem`*`( row, col, color )`: Set the given pixel to the given RGB `color`. The pixel coordinates must be within the valid range.

4.5 Color images can also be stored using three separate color channels in which the values of each color component is stored in a separate data structure. Implement a new version of the Color Image ADT using three 1-D arrays to store the red, green, and blue components of each pixel. Apply the row-major formula from Section 3.3 to map a specific pixel given by (`row`, `col`) to an entry in the 1-D arrays.

4.6 A color image can be easily converted to a grayscale image by converting each pixel of the color image, specified by the three components (R, G, B), to a grayscale value using the formula

```
gray = round( 0.299 * R + 0.587 * G + 0.114 * B )
```

The proportions applied to each color component in the formula corresponds to the levels of sensitivity with which humans see each of the three primary colors: red, green and blue. Note the result from the equation must be converted capped to an integer in the range $[0 \ldots 255]$. Use the equation and implement the function

```
def colorToGrayscale( colorImg ):
```

which accepts a `ColorImage` object as an argument and creates and returns a new `GrayscaleImage` that is the grayscale version of the given color image.