3
Principles of Algorithm Design

Why do we study algorithm design? There are, of course, many reasons, and our motivations for learning something is very much dependent on our own circumstances. There are, without a doubt, important professional reasons for being interested in algorithm design. Algorithms are the foundation of all computing. We can think of a computer as being a piece of hardware, with a hard drive, memory chips, processors, and so on. However, the essential component, the thing that, if missing, would render modern technology impossible, is algorithms. Let's learn more about it in the upcoming sections.

In this chapter, we will look at the following topics:

- An introduction to algorithms
- Recursion and backtracking
- Big O notation

Technical requirements

We will need to install the `matplotlib` library with Python to plot the diagram in this chapter.

It can be installed on Ubuntu/Linux by running the following commands on the terminal:

```
python3 -mpip install matplotlib
```

You can also use the following:

```
sudo apt-get install python3-matplotlib
```

To install `matplotlib` on Windows:

If Python is already installed on the Windows operating system, `matplotlib` can be obtained from the following link to install it on Windows: `https://github.com/matplotlib/matplotlib/downloads` or `https://matplotlib.org`.

Code files for this chapter can be found at: `https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Second-Edition/tree/master/Chapter03`.

An introduction to algorithms

The theoretical foundation of algorithms, in the form of the Turing machine, was established several decades before digital logic circuits could actually implement such a machine. The Turing machine is essentially a mathematical model that, using a predefined set of rules, translates a set of inputs into a set of outputs. The first implementations of Turing machines were mechanical and the next generation may likely see digital logic circuits replaced by quantum circuits or something similar. Regardless of the platform, algorithms play a central predominant role.

Another aspect is the effect algorithms have on technological innovation. As an obvious example, consider the page rank search algorithm, a variation of which the Google Search engine is based on. Using this and similar algorithms allows researchers, scientists, technicians, and others to quickly search through vast amounts of information extremely quickly. This has a massive effect on the rate at which new research can be carried out, new discoveries made, and new innovative technologies developed. An algorithm is a sequential set of instructions to execute a particular task. They are very important, as we can break a complex problem into a smaller one to prepare simple steps to execute a big problem—that is the most important part of algorithms. A good algorithm is key for an efficient program to solve a specific problem. The study of algorithms is also important because it trains us to think very specifically about certain problems. It can help to increase our problem-solving abilities by isolating the components of a problem and defining relationships between these components. In summary, there are some important reasons for studying algorithms:

- They are essential for computer science and *intelligent* systems
- They are important in many other domains (computational biology, economics, ecology, communications, ecology, physics, and so on)
- They play a role in technology innovation
- They improve problem-solving and analytical thinking

There are mainly two important aspects to solve a given problem. Firstly, we need an efficient mechanism to store, manage, and retrieve the data, which is important to solve a problem (this comes under data structures); secondly, we require an efficient algorithm which is a finite set of instructions to solve that problem. Thus, the study of data structures and algorithms is key to solving any problem using computer programs. An efficient algorithm should have the following characteristics:

- It should be as specific as possible
- It should have each instruction properly defined
- There should not be any ambiguous instruction
- All the instructions of the algorithm should be executable in a finite amount of time and in a finite number of steps
- It should have clear input and output to solve the problem
- Each instruction of the algorithm should be important in solving the given problem

Algorithms, in their simplest form, are just a sequence of actions—a list of instructions. It may just be a linear construct of the form do *x*, then do *y*, then do *z*, then finish. However, to make things more useful we add clauses to the effect of do *x* then do *y*; in Python, these are if-else statements. Here, the future course of action is dependent on some conditions; say the state of a data structure. To this, we also add the operation, iteration, the while, and the for statements. Expanding our algorithmic literacy further, we add recursion. Recursion can often achieve the same results as iteration, however, they are fundamentally different. A recursive function calls itself, applying the same function to progressively smaller inputs. The input of any recursive step is the output of the previous recursive step.

Algorithm design paradigms

In general, we can discern three broad approaches to algorithm design. They are:

- Divide and conquer
- Greedy algorithms
- Dynamic programming

As the name suggests, the divide and conquer paradigm involves breaking a problem into smaller simple sub-problems, and then solving these sub-problems, and finally, combining the results to obtain a global optimal solution. This is a very common and natural problem-solving technique, and is, arguably, the most commonly used approach to algorithm design. For example, merge sort is an algorithm to sort a list of n natural numbers increasingly.

In this algorithm, we divide the list iteratively in equal parts until each sub-list contains one element, and then we combine these sub-lists to create a new list in a sorted order. We will be discussing merge sort in more detail later in this section/chapter.

Some examples of divide and conquer algorithm paradigms are as follows:

- Binary search
- Merge sort
- Quick sort
- Karatsuba algorithm for fast multiplication
- Strassen's matrix multiplication
- Closest pair of points

Greedy algorithms often involve optimization and combinatorial problems. In greedy algorithms, the objective is to obtain the best optimum solution from many possible solutions in each step, and we try to get the local optimum solution which may eventually lead us to obtain the overall optimum solution. Generally, greedy algorithms are used for optimization problems. Here are many popular standard problems where we can use greedy algorithms to obtain the optimum solution:

- Kruskal's minimum spanning tree
- Dijkstra's shortest path
- Knapsack problem
- Prim's minimal spanning tree algorithm
- Travelling salesman problem

Greedy algorithms often involve optimization and combinatorial problems; the classic example is to apply the greedy algorithm to the traveling salesperson problem, where a greedy approach always chooses the closest destination first. This shortest-path strategy involves finding the best solution to a local problem in the hope that this will lead to a global solution.

Another classic example is to apply the greedy algorithm to the traveling salesperson problem; it is an NP-hard problem. In this problem, a greedy approach always chooses the closest unvisited city first from the current city; in this way, we are not sure that we get the best solution, but we surely get an optimal solution. This shortest-path strategy involves finding the best solution to a local problem in the hope that this will lead to a global solution.

The dynamic programming approach is useful when our sub-problems overlap. This is different from divide and conquer. Rather than breaking our problem into independent sub-problems, with dynamic programming, intermediate results are cached and can be used in subsequent operations. Like divide and conquer, it uses recursion; however, dynamic programming allows us to compare results at different stages. This can have a performance advantage over the divide and conquer for some problems because it is often quicker to retrieve a previously calculated result from memory rather than having to recalculate it. Dynamic programming also uses recursion to solve the problems. For example, the matrix chain multiplication problem can be solved using dynamic programming. The matrix chain multiplication problem determines the best effective way to multiply the matrices when a sequence of matrices is given, it finds the order of multiplication that requires the minimum number of operations.

For example, let's look at three matrices—*P*, *Q*, and *R*. To compute the multiplication of these three matrices, we have many possible choices (because the matrix multiplication is associative), such as *(PQ)R = P(QR)*. So, if the sizes of these matrices are—*P* is a 20 × 30, *Q* is 30 × 45, *R* is 45 x 50, then, the number of multiplications for *(PQ)R* and *P(QR)* will be:

- *(PQ)R* = 20 x 30 x 45 + 20 x 45 x 50 = 72,000
- *P(QR)* = 20 x 30 x 50 + 30 x 45 x 50 = 97,500

It can be observed from this example that if we multiply using the first option, then we would need 72,000 multiplications, which is less when compared to the second option/ This is shown in the following code:

```
def MatrixChain(mat, i, j):
    if i == j:
        return 0
    minimum_computations = sys.maxsize
    for k in range(i, j):
        count = (MatrixChain(mat, i, k) + MatrixChain(mat, k+1, j)+
mat[i-1] * mat[k] * mat[j])
        if count < minimum_computations:
              minimum_computations= count;
        return minimum_computations;

matrix_sizes = [20, 30, 45, 50];
print("Minimum multiplications are", MatrixChain(matrix_sizes , 1,
len(matrix_sizes)-1));

#prints 72000
```

`Chapter 13`, *Design Techniques and Strategies*, presents a more detailed discussion on the algorithm design strategy.

Recursion and backtracking

Recursion is particularly useful for divide and conquer problems; however, it can be difficult to understand exactly what is happening, since each recursive call is itself spinning off other recursive calls. A recursive function can be in an infinite loop, therefore, it is required that each recursive function adhere to some properties. At the core of a recursive function are two types of cases:

- **Base cases**: These tell the recursion when to terminate, meaning the recursion will be stopped once the base condition is met
- **Recursive cases**: The function calls itself and we progress towards achieving the base criteria

A simple problem that naturally lends itself to a recursive solution is calculating factorials. The recursive factorial algorithm defines two cases: the base case when *n* is zero (the terminating condition), and the recursive case when *n* is greater than zero (the call of the function itself). A typical implementation is the following:

```
def factorial(n):
    # test for a base case
    if  n==0:
        return 1
        #make a calculation and a recursive call
    else:
        f= n*factorial(n-1)
    print(f)
    return(f)

factorial(4)
```

To calculate the factorial of `4`, we require four recursive calls plus the initial parent call. On each recursion, a copy of the method variables is stored in memory. Once the method returns it is removed from memory. The following is a way we can visualize this process:

It may not necessarily be clear if recursion or iteration is a better solution to a particular problem; after all, they both repeat a series of operations and both are very well-suited to divide and conquer approaches and to algorithm design. Iteration churns away until the problem is done with. Recursion breaks the problem down into smaller and smaller chunks and then combines the results. Iteration is often easier for programmers, because control stays local to a loop, whereas recursion can more closely represent mathematical concepts such as factorials. Recursive calls are stored in memory, whereas iterations are not. This creates a trade-off between processor cycles and memory usage, so choosing which one to use may depend on whether the task is processor or memory intensive. The following table outlines the key differences between recursion and iteration:

Recursion	**Iteration**
The function calls itself.	A set of instructions are executed repeatedly in the loop.
It stops when the termination condition is met.	It stops execution when the loop condition is met.
Infinite recursive calls may give an error related to stack overflow.	An infinite iteration will run indefinitely until the hardware is powered.
Each recursive call needs memory space.	Each iteration does not require memory storage.
The code size, in general, is comparatively smaller.	The code size, in general, is comparatively smaller.
Recursion is generally slower than iteration.	It is faster as it does not require a stack.

Backtracking

Backtracking is a form of recursion that is particularly useful for types of problems such as traversing tree structures, where we are presented with a number of options for each node, from which we must choose one. Subsequently, we are presented with a different set of options, and depending on the series of choices made, either a goal state or a dead end is reached. If it is the latter, we must backtrack to a previous node and traverse a different branch. Backtracking is a divide and conquer method for exhaustive searching. Importantly, backtracking **prunes** branches that cannot give a result.

An example of backtracking is given next. Here, we have used a recursive approach to generate all the possible arrangements of a given string, `s`, of a given length, `n`:

```
def bitStr(n,s):
 if n==1: return s
 return [digit + bits for digit in bitStr(1,s) for bits in bitStr(n-1,s)]

print(bitStr(3,'abc'))
```

This generates the following output:

```
['aaa', 'aab', 'aac', 'aba', 'abb', 'abc', 'aca', 'acb', 'acc', 'baa', 'bab', 'bac', 'bba',
'bbb', 'bbc', 'bca', 'bcb', 'bcc', 'caa', 'cab', 'cac', 'cba', 'cbb', 'cbc', 'cca', 'ccb', 'ccc']
```

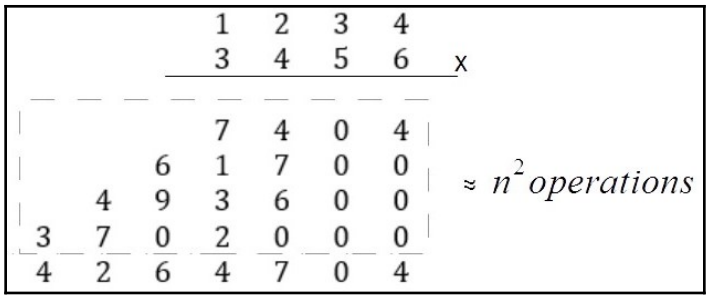
Notice the double list compression and the two recursive calls within this comprehension. This recursively concatenates each element of the initial sequence, returned when n =1, with each element of the string generated in the previous recursive call. In this sense, it is *backtracking* to uncover previously ungenerated combinations. The final string that is returned is all n letter combinations of the initial string.

Divide and conquer – long multiplication

For recursion to be more than just a clever trick, we need to understand how to compare it to other approaches, such as iteration, and to understand when its use will lead to a faster algorithm. An iterative algorithm that we are all familiar with is the procedure we learned in primary math classes, and is used to multiply two large numbers. That is long multiplication. If you remember, long multiplication involved iterative multiplying and carry operations followed by a shifting and addition operation.

Our aim here is to examine ways to measure how efficient this procedure is and attempt to answer the question—is this the most efficient procedure we can use for multiplying two large numbers together?

In the following diagram, we can see that multiplying two four-digit numbers together requires 16 multiplication operations, and we can generalize and say that an n digit number requires, approximately, n^2 multiplication operations:



This method of analyzing algorithms, in terms of the number of computational primitives such as multiplication and addition, is important because it gives us a way to understand the relationship between the time it takes to complete a certain computation and the size of the input to that computation. In particular, we want to know what happens when the input, the number of digits, n, is very large. This topic, called **asymptotic analysis**, or **time complexity**, is essential to our study of algorithms and we will revisit it often during this chapter and the rest of this book.

The recursive approach

It turns out that in the case of long multiplication the answer is yes, there are in fact several algorithms for multiplying large numbers that require less operations. One of the most well-known alternatives to long multiplication is the **Karatsuba algorithm**, first published in 1962. This takes a fundamentally different approach: rather than iteratively multiplying single-digit numbers, it recursively carries out multiplication operations on progressively smaller inputs. Recursive programs call themselves on smaller subsets of the input. The first step in building a recursive algorithm is to decompose a large number into several smaller numbers. The most natural way to do this is to simply split the number into two halves, the first half of most-significant digits, and a second half of least-significant digits. For example, our four-digit number, 2345, becomes a pair of two-digit numbers, 23 and 45. We can write a more general decomposition of any two n digit numbers, x, and y using the following, where m is any positive integer less than n:

$$x = 10^m a + b$$

$$y = 10^m c + d$$

So now we can rewrite our multiplication problem x, y as follows:

$$(10^m a + b)(10^m c + d)$$

When we expand, we get the following:

$$10^{2m}ac + 10^{m}(ad + bc) + bd$$

More conveniently, we can write it like this (equation 3.1):

$$x \times y = 10^{2m}z_2 + 10^{m}z_1 + z_0 \qquad \text{... (3.1)}$$

Where:

$$z_2 = ac; z_1 = ad + bc; z_0 = bd$$

It should be pointed out that this suggests a recursive approach to multiplying two numbers since this procedure does itself involve multiplication. Specifically, the products *ac*, *ad*, *bc*, and *bd* all involve numbers smaller than the input number and so it is conceivable that we could apply the same operation as a partial solution to the overall problem. This algorithm, so far, consists of four recursive multiplication steps and it is not immediately clear if it will be faster than the classic long multiplication approach.

What we have discussed so far in regards to the recursive approach to multiplication, has been well-known to mathematicians since the late nineteenth century. The Karatsuba algorithm improves on this by making the following observation. We really only need to know three quantities: z_2= *ac*, z_1=*ad* +*bc*, and z_0= *bd* to solve equation 3.1. We need to know the values of *a*, *b*, *c*, and *d* only in so far as they contribute to the overall sum and products involved in calculating the quantities z_2, z_1, and z_0. This suggests the possibility that perhaps we can reduce the number of recursive steps. It turns out that this is indeed the situation.

Since the products *ac* and *bd* are already in their simplest form, it seems unlikely that we can eliminate these calculations. We can, however, make the following observation:

$$(a + b)(c + d) = ac + bd + ad + bc$$

When we subtract the quantities *ac* and *bd*, which we have calculated in the previous recursive step, we get the quantity we need, namely (*ad* + *bc*):

$$ac + bd + ad + bc - ac - bd = ad + bc$$

This shows that we can indeed compute the sum of $ad + bc$ without separately computing each of the individual quantities. In summary, we can improve on equation 3.1 by reducing four recursive steps to three. These three steps are as follows:

1. Recursively calculate ac
2. Recursively calculate bd
3. Recursively calculate $(a + b)(c + d)$ and subtract ac and bd

The following example shows a Python implementation of the Karatsuba algorithm. In the following code, initially, we see if any one of the given numbers is less than 10, then there is no need to run recursive functions. Next, we identify the number of digits in the larger value, and add one if the number of digits is odd. Finally, we recursively call the function three times to calculate ac, bd, and $(a + d)(c + d)$. The following code prints the multiplication of any two digits; for example, it prints `4264704` for the multiplication of `1234` and `3456`. The implementation of the Karatsuba algorithm is:

```
from math import log10
def karatsuba(x,y):

    #The base case for recursion
    if x<10 or y<10:
        return x*y

    #sets n, the number of digits in the highest input number
    n=max(int(log10(x)+1), int(log10(y)+1))

    #rounds up n/2
    n_2 = int(math.ceil(n/2.0))
    #adds 1 if n is uneven
    n = n if n%2 == 0  else n+1
    #splits the input numbers
    a, b = divmod(x, 10**n_2)
    c, d = divmod(y,10**n_2)
    #applies the three recursive steps
    ac = karatsuba(a,c)
    bd = karatsuba(b,d)
    ad_bc = karatsuba((a+b),(c+d))-ac-bd

    #performs the multiplication
    return (((10**n)*ac)+bd+((10**n_2)*(ad_bc)))


t= karatsuba(1234,3456)
print(t)

# outputs - 4264704
```

Runtime analysis

The performance of an algorithm is generally measured by the size of its input data (**n**) and the time and the memory space used by the algorithm. **Time** required is measured by the key operations to be performed by the algorithm (such as comparison operations), whereas the space requirements of an algorithm is measured by the storage needed to store the variables, constants, and instructions during the execution of the program. The space requirements of an algorithm may also change dynamically during execution as it depends on variable size, which is to be decided at runtime, such as dynamic memory allocation, memory stacks, and so on.

The running time required by an algorithm depends on the input size; as the input size (**n**) increases, the runtime also increases. For example, a sorting algorithm will have more running time to sort the list of input size 5,000 as compared to the other list of input size 50. Therefore, it is clear that to compute the time complexity, the input size is important. Further, for a specific input, the running time depends on the key operations to be executed in the algorithm. For example, the key operation for a sorting algorithm is a **comparison operation** that will take most of the time as compared to assignment or any other operation. The more is the number of key operations to be executed, the longer it will take to run the algorithm.

It should be noted that an important aspect to algorithm design is gauging the efficiency both in terms of space (memory) and time (number of operations). It should be mentioned that an identical metric is used to measure an algorithm's memory performance. There are a number of ways we could, conceivably, measure runtime and probably the most obvious way is to simply measure the total time taken by the algorithm. The major problem with this approach is that the time taken for an algorithm to run is very much dependent on the hardware it is run on. A platform-independent way to gauge an algorithm's runtime is to count the number of operations involved. However, this is also problematic as there is no definitive way to quantify an operation. This is dependent on the programming language, the coding style, and how we decide to count operations. We can use this idea, though, of counting operations, if we combine it with the expectation that as the size of the input increases the runtime will increase in a specific way. That is, there is a mathematical relationship between n, the size of the input, and the time it takes for the algorithm to run. There are essentially three things that characterize an algorithm's runtime performance; these can be described as follows:

- Worst-case complexity is the upper-bound complexity; it is the maximum running time required for an algorithm to execute. In this case, the key operations would be executed the maximum number of times.

- Best-case complexity is the lower-bound complexity; it is the minimum running time required for an algorithm to execute. In this case, the key operations would be executed the minimum number of times.
- Average-case complexity is the average running time required for an algorithm to execute.

Worst-case analysis is useful because it gives us a tight upper bound that our algorithm is guaranteed not to exceed. Ignoring small constant factors, and lower-order terms, is really just about ignoring the things that, at large values of the input size, n, do not contribute, in a large degree, to the overall run time. Not only does this make our work mathematically easier, but it also allows us to focus on the things that are having the most impact on performance.

We saw with the Karatsuba algorithm that the number of multiplication operations increased to the square of the size, n, of the input. If we have a four-digit number the number of multiplication operations is 16; an eight-digit number requires 64 operations. Typically, though, we are not really interested in the behavior of an algorithm at small values of n, so we most often ignore factors that increase at slower rates, say linearly with n. This is because at high values of n, the operations that increase the fastest as we increase n will dominate.

We will explain this in more detail with an example: the merge sort algorithm. Sorting is the subject of `Chapter 10`, *Sorting*, however, as a precursor and as a useful way to learn about runtime performance, we will introduce merge sort here.

The merge sort algorithm is a classic algorithm developed over 60 years ago. It is still used widely in many of the most popular sorting libraries. It is relatively simple and efficient. It is a recursive algorithm that uses a divide and conquer approach. This involves breaking the problem into smaller sub-problems, recursively solving them, and then somehow combining the results. Merge sort is one of the most obvious demonstrations of the divide and conquer paradigm.

The merge sort algorithm consists of three simple steps:

1. Recursively sort the left half of the input array
2. Recursively sort the right half of the input array
3. Merge two sorted sub-arrays into one

A typical problem is sorting a list of numbers into a numerical order. Merge sort works by splitting the input into two halves and working on each half in parallel. We can illustrate this process schematically with the following diagram:

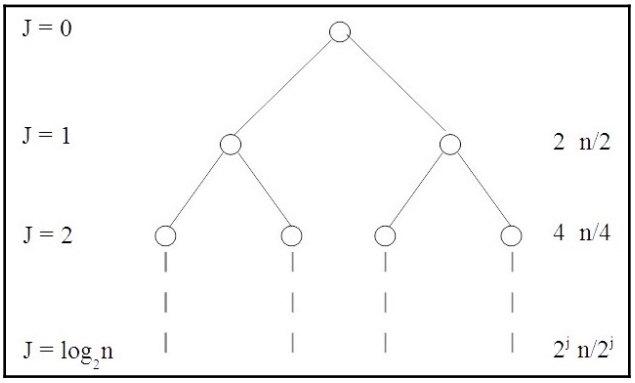
Here is the Python code for the merge sort algorithm:

```
def mergeSort(A):
#base case if the input array is one or zero just return.
if len(A) > 1:
    # splitting input array
    print('splitting ', A )
    mid=len(A)//2
    left=A[:mid]
    right=A[mid:]
    #recursive calls to mergeSort for left and right subarrays
    mergeSort(left)
    mergeSort(right)
    #initalizes pointers for left(i) right(j) and output array (k)
    #3 initalization operations
    i = j = k = 0
    #Traverse and merges the sorted arrays
    while i < len(left) and j < len(right):
    #if left < right comparison operation
        if left[i] < right[j]:
        #if left < right Assignment operation
            A[k] = left[i]
            i=i+1
        else:
            #if right <= left assignment
            A[k]=right[j]
            j=j+1
            k=k+1
    while i< len(left):
    #Assignment operation
        A[k] = left[i]
        i=i+1
        k=k+1

    while j< len(right):
    # Assignment operation
        A[k] = right[j]
        j=j+1
        k=k+1

print('merging',A)
return(A)
```

We run this program for the following results:

```
In [2]: mergeSort([356,97,846,215])
splitting  [356, 97, 846, 215]
splitting  [356, 97]
merging  [356]
merging  [97]
merging  [97, 356]
splitting  [846, 215]
merging  [846]
merging  [215]
merging  [215, 846]
merging  [97, 215, 356, 846]
Out[2]: [97, 215, 356, 846]
```

The problem that we are interested in is how we determine the runtime performance, that is, what is the rate of growth in the time it takes for the algorithm to complete relative to the size of n? To understand this a bit better, we can map each recursive call onto a tree structure. Each node in the tree is a recursive call working on progressively smaller sub-problems:



Each invocation of merge sort subsequently creates two recursive calls, so we can represent this with a binary tree. Each of the child nodes receives a subset of the input. Ultimately, we want to know the total time it takes for the algorithm to complete relative to the size of n. To begin with, we can calculate the amount of work and the number of operations at each level of the tree.

Focusing on the runtime analysis, at level one, the problem is split into two $n/2$ sub-problems; at level two, there are four $n/4$ subproblems, and so on. The question is, when does the recursion bottom out, that is, when does it reach its base case? This is simply when the array is either zero or one.

The number of recursive levels is exactly the number of times you need to divide n by two until you get a number that is at most one. This is precisely the definition of log2. Since we are counting the initial recursive call as level zero, the total number of levels is $\log_2 n + 1$.

Let's just pause to refine our definitions. So far, we have been describing the number of elements in our input by the letter n. This refers to the number of elements in the first level of the recursion, that is, the length of the initial input. We are going to need to differentiate between the size of the input at subsequent recursive levels. For this, we will use the letter m or specifically m_j for the length of the input at recursive level j.

Also, there are a few details we have overlooked, and I am sure you are beginning to wonder about. For example, what happens when $m/2$ is not an integer, or when we have duplicates in our input array? It turns out that this does not have an important impact on our analysis here; we will revisit some of the finer details of the merge sort algorithm in `Chapter 12`, *Design Techniques and Strategies*.

The advantage of using a recursion tree to analyze algorithms is that we can calculate the work done at each level of the recursion. How we define this work is simply by the total number of operations and this, of course, is related to the size of the input. It is important to measure and compare the performance of algorithms in a platform-independent way. The actual runtime will, of course, be dependent on the hardware on which it is run. Counting the number of operations is important because it gives us a metric that is directly related to an algorithm's performance, independent of the platform.

In general, since each invocation of merge sort is making two recursive calls, the number of calls is doubling at each level. At the same time, each of these calls is working on an input that is half of its parents. We can formalize this and say that for level j, where j is an integer $0, 1, 2 \ldots \log_2 n$, there are two sub-problems each of size $n/2^j$.

To calculate the total number of operations, we need to know the number of operations encompassed by a single merge of two sub-arrays. Let's count the number of operations in the previous Python code. What we are interested in is all the code after the two recursive calls have been made. Firstly, we have the three assignment operations. This is followed by three `while` loops. In the first loop, we have an if-else statement and within each of our two operations, a comparison followed by an assignment. Since there are only one of these sets of operations within the if-else statements, we can count this block of code as two operations carried out m times. This is followed by two `while` loops with an assignment operation each. This makes a total of $4m + 3$ operations for each recursion of merge sort.

Since m must be at least one, the upper bound for the number of operations is $7m$. It has to be said that this has no pretence at being an exact number. We could, of course, decide to count operations in a different way. We have not counted the increment operations or any of the housekeeping operations; however, this is not so important as we are more concerned with the rate of growth of the runtime with respect to n at high values of n.

This may seem a little daunting since each call of a recursive call itself spins off more recursive calls, and seemingly explodes exponentially. The key fact that makes this manageable is that as the number of recursive calls doubles, the size of each subproblem halves. These two opposing forces cancel out nicely, as we can demonstrate.

To calculate the maximum number of operations at each level of the recursion tree we simply multiply the number of subproblems by the number of operations in each subproblem as follows:

$$2^j \times 7(n/2^j) = 7n$$

Importantly, this shows that, because the 2^j cancels out the number of operations at each level is independent of the level. This gives us an upper bound to the number of operations carried out on each level, in this example, $7n$. It should be pointed out that this includes the number of operations performed by each recursive call on that level, not the recursive calls made on subsequent levels. This shows that the work is done, as the number of recursive calls doubles with each level, and is exactly counterbalanced by the fact that the input size for each sub-problem is halved.

To find the total number of operations for a complete merge sort, we simply multiply the number of operations on each level by the number of levels. This gives us the following:

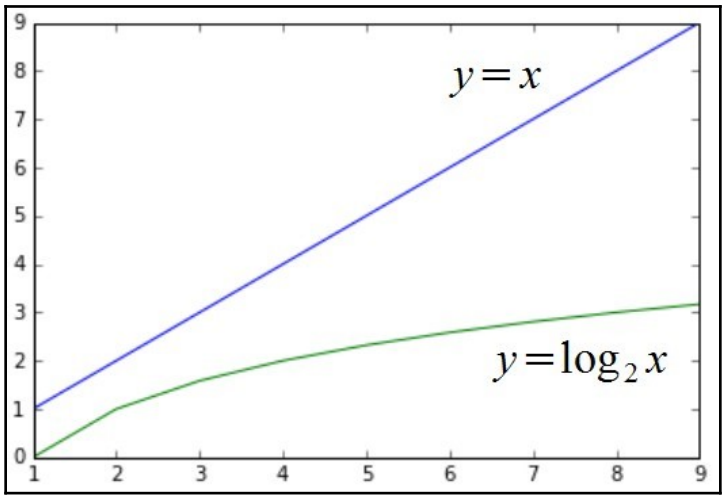
$$7n(log_2 n + 1)$$

When we expand this out, we get the following:

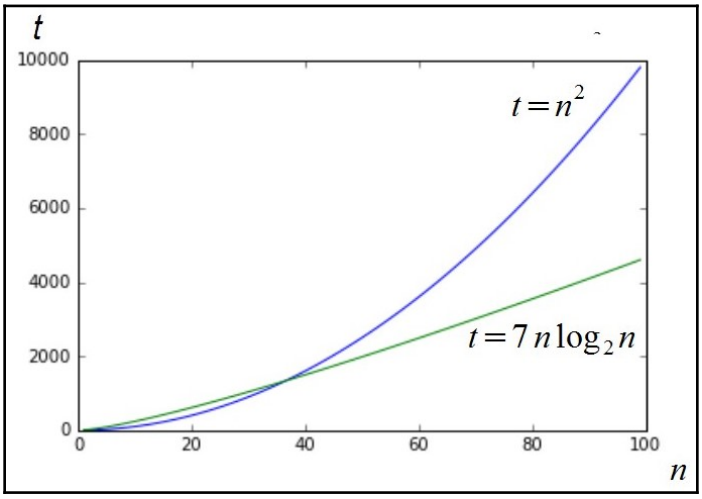
$$7nlog_2 n + 7$$

The key point to take from this is that there is a logarithmic component to the relationship between the size of the input and the total running time. If you remember from school mathematics, the distinguishing characteristic of the logarithm function is that it flattens off very quickly. As an input variable, x increases in size; the output variable y increases by smaller and smaller amounts.

For example, compare the log function to a linear function:



In the previous example, multiplying the $n\log_2 n$ component and comparing it to n^2:



Notice how for very low values of n, the time to complete, t, is actually lower for an algorithm that runs in n2 time. However, for values above about 40, the log function begins to dominate, flattening the output until, at the comparatively moderate size $n = 100$, the performance is more than twice than that of an algorithm running in n^2 time. Notice also that the disappearance of the constant factor, + 7, is irrelevant at high values of n.

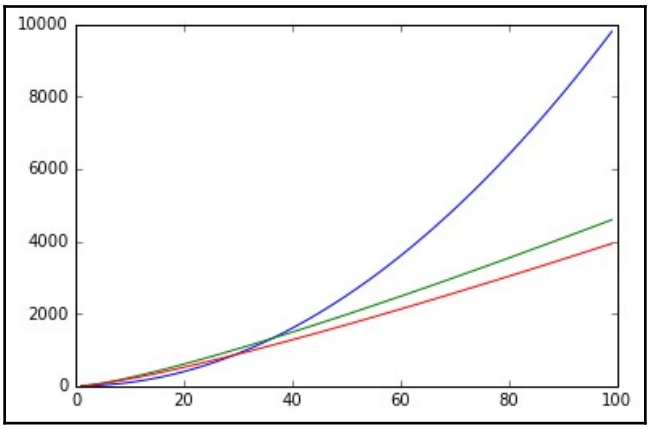
The code used to generate these graphs is as follows:

```
import matplotlib.pyplotasplt
import math
x = list(range(1,100))
l=[]; l2=[]; a=1
plt.plot(x, [y*y for y in x])
plt.plot(x, [(7*y)*math.log(y,2) for y in x])
plt.show()
```

You will need to install the `matplotlib` library, if it is not installed already, for this to work. Details can be found at the following address; I encourage you to experiment with this list comprehension expression used to generate the plots. For example, we could add the following `plot` statement:

```
plt.plot(x, [(6*y)* math.log(y, 2) for y in x])
```

This gives the following output:



The preceding graph shows the difference between counting six operations or seven operations. We can see how the two cases diverge, and this is important when we are talking about the specifics of an application. However, what we are more interested in here is a way to characterize growth rates. We are not so much concerned with the absolute values, but how these values change as we increase n. In this way, we can see that the two lower curves have similar growth rates when compared to the top (x^2) curve. We say that these two lower curves have the same **complexity class**. This is a way to understand and describe different runtime behaviors. We will formalize this performance metric in the next section.

Asymptotic analysis

Asymptotic analysis of an algorithm refers to the computation of the running time of the algorithm. To determine which algorithm is better, given two algorithms, a simple approach can be to run both the programs, and the algorithm that takes the least time to execute for a given input is better than the other. However, it is possible that for a specific input, one algorithm performs better than other, whereas for any other input value that the algorithm may perform worse.

In asymptotic analysis, we compare two algorithms with respect to input size rather than the actual runtime, and we measure how the time taken increases with the increase in input size. This is depicted with the following code:

```
# Linear search program to search an element, return the index position of
the #array
def searching(search_arr, x):
    for i in range(len(search_arr)):
        if search_arr [i] == x:
                return i
    return -1

search_ar= [3, 4, 1, 6, 14]
x=4

searching(search_ar, x)
print("Index position for the element x is :",searching(search_ar, x))

#outputs index position of the element x that is - 1
```

Assuming that the size of the array is `n`, and *T(n)* is the total number of key operations required to perform a linear search, the key operation in this example is the comparison. Let's consider the linear search as an example to understand the worst case, average-case, and best-case complexity:

- **Worst-case analysis**: We consider the upper-bound running time, that is, the maximum time to be taken by the algorithm. In the linear search, the worst case happens when the element to be searched is found in the last comparison or not found in the list. In this case, there will be a maximum number of comparisons and that will be the total number of elements in the array. Therefore, the worst-case time complexity is Θ(n).

- **Average-case analysis**: In this analysis, we consider all the possible cases where the element can be found in the list, and then, we compute the average running time complexity. For example, in the linear search, the number of comparisons at all the positions would be *1* if the element to be searched was found at *0th* index, and similarly, the number of comparisons would be 2, 3, and so forth, up to *n* respectively for elements found at *1*, *2*, *3*, … *(n-1)* index positions. Thus the average time complexity can defined as `average-case complexity= (1+2+3...n)/n = n(n+1)/2`.
- **Best-case analysis**: Best-case running time complexity is the minimum time needed for an algorithm to run; it is the lower-bound running time. In a linear search, the best case would be if the element to be searched is found in the first comparison. In this example, it is clear that the best-case time complexity is not dependent upon how long the list is. So, the best-case time complexity would be *Θ(1)*.

Generally, we use worst-case analysis to analyze an algorithm as it provides us with the upper bound on the running time, whereas best-case anaylsis is the least important as it provides us with the lower bound—that is, a minimum time required for an algorithm. Furthermore, the computation of average-case analysis is very difficult.

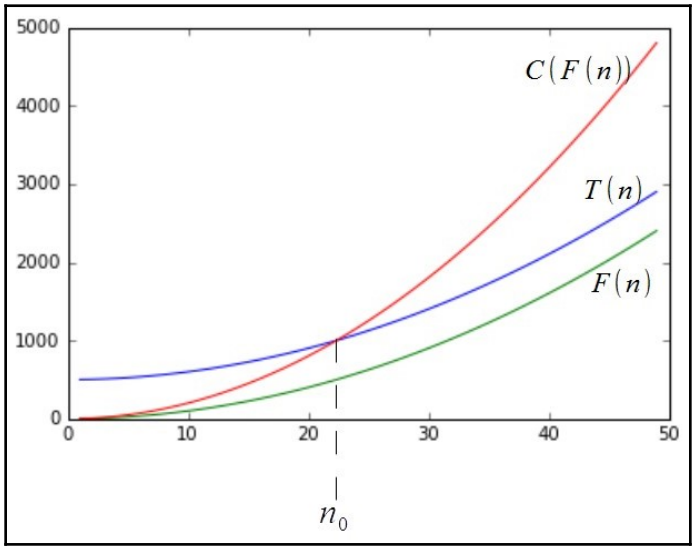
To calculate each of these, we need to know the upper and lower bounds. We have looked at a way to represent an algorithm's runtime using mathematical expressions, essentially adding and multiplying operations. To use asymptotic analysis, we simply create two expressions, one each for the best and worst cases.

Big O notation

The letter O in big *O* notation stands for order, in recognition that rates of growth are defined as the order of a function. It measures the worst-case running time complexity, that is, the maximum time to be taken by the algorithm. We say that one function $T(n)$ is a big O of another function, $F(n)$, and we define this as follows:

$$T(n) = O(F(n) \ \ iff\ there\ exists\ constants,\ n_0\ and\ c\ such\ that:\ T(n) <= c(F(n))\ for\ all\ n >= n_0)$$

The function, $g(n)$, of the input size, n, is based on the observation that for all sufficiently large values of n, $g(n)$ is bounded above by a constant multiple of $f(n)$. The objective is to find the smallest rate of growth that is less than or equal to $f(n)$. We only care what happens at higher values of n. The variable *n0* represents the threshold below which the rate of growth is not important. The function *T(n)* represents the **tight upper bound** F(n). In the following plot, we can see that $T(n) = n^2 + 500 = O(n^2)$, with $C = 2$ and n_0 being approximately 23:



You will also see the notation $f(n) = O(g(n))$. This describes the fact that $O(g(n))$ is really a set of functions that includes all functions with the same or smaller rates of growth than f(n). For example, $O(n^2)$ also includes the functions *O(n)*, *O(nlogn)*, and so on. Let's consider another example.

The big O time complexity for the function `f(x)= 19n log2n +56` is *O(nlogn)*.

In the following table, we list the most common growth rates in order from lowest to highest. We sometimes call these growth rates the **time complexity** of a function, or the complexity class of a function:

Complexity class	**Name**	**Example operations**
O(1)	Constant	append, get item, set item.
O(logn)	Logarithmic	Finding an element in a sorted array.
O(n)	Linear	copy, insert, delete, iteration.

nLogn	Linear-logarithmic	Sort a list, merge-sort.
n^2	Quadratic	Find the shortest path between two nodes in a graph. Nested loops.
n^3	Cubic	Matrix multiplication.
2^n	Exponential	**Towers of Hanoi** problem, backtracking.

Composing complexity classes

Normally, we need to find the total running time of a number of basic operations. It turns out that we can combine the complexity classes of simple operations to find the complexity class of more complex, combined operations. The goal is to analyze the combined statements in a function or method to understand the total time complexity of executing several operations. The simplest way to combine two complexity classes is to add them. This occurs when we have two sequential operations. For example, consider the two operations of inserting an element into a list and then sorting that list. We can see that inserting an item occurs in *O(n)* time and sorting is in *O(nlogn)* time. We can write the total time complexity as *O(n + nlogn)*, that is, we bring the two functions inside the *O(...)*. We are only interested in the highest-order term, so this leaves us with just *O(nlogn)*.

If we repeat an operation, for example, in a `while` loop, then we multiply the complexity class by the number of times the operation is carried out. If an operation with time complexity *O(f(n))* is repeated *O(n)* times then we multiply the two complexities:

$$O(f(n) * O(n)) = O(nf(n))$$

For example, suppose the function `f(...)` has a time complexity of $O(n^2)$ and it is executed *n* times in a `while` loop, as follows:

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

The time complexity of this loop then becomes $O(n^2) * O(n) = O(n * n^2) = O(n^3)$. Here we are simply multiplying the time complexity of the operation by the number of times this operation executes. The running time of a loop is at most the running time of the statements inside the loop multiplied by the number of iterations. A single nested loop, that is, one loop nested inside another loop, will run in *n*2 time assuming both loops run `n` times, as demonstrated in the following example:

```
for i in range(0,n):
    for j in range(0,n)
            #statements
```

Each statement is a constant, *c*, executed *nn* times, so we can express the running time as the following:

$$cnn = cn2 = O(n2).$$

For consecutive statements within nested loops, we add the time complexities of each statement and multiply by the number of times the statement executed, for example:

```
n=500  #c0
#executes n times
for i in range(0,n):
    print(i)    #c1
   #executes n times
for i in range(0,n):
#executes n times
    for j in range(0,n):
            print(j)  #c2
```

This can be written as $c_0 + c_1 n + cn^2 = O(n^2)$.

We can define (base 2) logarithmic complexity, reducing the size of the problem by ½, in constant time. For example, consider the following snippet:

```
i=1
while i<=n:
    i=i*2
    print(i)
```

Notice that i is doubling on each iteration; if we run this with $n = 10$ we see that it prints out four numbers: 2, 4, 8, and 16. If we double n we see it prints out five numbers. With each subsequent doubling of n, the number of iterations is only increased by one. If we assume k iterations, we can write this as follows:

$$log_2(2^k) = log_2 n$$

$$klog_2 2 = log_2 n$$

$$k = log(n)$$

From this, we can conclude that the total time = $\boldsymbol{O}(log(n))$.

Although big O is the most used notation involved in asymptotic analysis, there are two other related notations that should be briefly mentioned. They are Omega notation and Theta notation.

Omega notation (Ω)

Omega notation describes tight lower bound on algorithms, similar to big O notation which describes a tight upper bound. Omega notation computes the best-case running time complexity of the algorithm. It provides the highest rate of growth *T(n)* which is less than or equal to the given algorithm. It can be computed as follows:

$$T(n) = \Omega(F(n)\ \ iff\ there\ exists\ constants,\ n_0\ and\ c\ such\ that:\ 0 <= c(F(n)) <= T(n)\ for\ all\ n >= n_0)$$

Theta notation (Θ)

It is often the case where both the upper and lower bounds of a given function are the same and the purpose of Theta notation is to determine if this is the case. The definition is as follows:

$$T(n) = \theta(F(n)\ \ iff\ there\ exists\ constants,\ n_0\ and\ c_1\ and\ c_2\ such\ that:\ 0 <= c_1(F(n)) <= T(n) <= c2(F(n))\ for\ all\ n >= n_0)$$

Although Omega and Theta notations are required to completely describe growth rates, the most practically useful is big O notation and this is the one you will see most often.

Amortized analysis

Often we are not so interested in the time complexity of individual operations; we are more interested in the average running time of sequences of operations. This is called amortized analysis. It is different from average-case analysis, which we will discuss shortly, in that we make no assumptions regarding the data distribution of input values. It does, however, take into account the state change of data structures. For example, if a list is sorted, any subsequent find operations should be quicker. The amortized analysis considers the state change of data structures because it analyzes sequences of operations, rather than simply aggregating single operations.

Amortized analysis describes an upper bound on the runtime of the algorithm; it imposes an additional cost on each operation in the algorithm. The additional considered cost of a sequence may be cheaper as compared to the initial expensive operation.

When we have a small number of expensive operations, such as sorting, and lots of cheaper operations such as lookups, standard worst-case analysis can lead to overly pessimistic results, since it assumes that each lookup must compare each element in the list until a match is found. We should take into account that once we sort the list we can make subsequent find operations cheaper.

So far in our runtime analysis, we have assumed that the input data was completely random and have only looked at the effect the size of the input has on the runtime. There are two other common approaches to algorithm analysis; they are:

- Average-case analysis
- Benchmarking

Average-case analysis will find the average running time which is based on some assumptions regarding the relative frequencies of various input values. Using real-world data, or data that replicates the distribution of real-world data, is many times on a particular data distribution and the average running time is calculated.

Benchmarking is simply having an agreed set of typical inputs that are used to measure performance. Both benchmarking and average-time analysis rely on having some domain knowledge. We need to know what the typical or expected datasets are. Ultimately, we will try to find ways to improve performance by fine-tuning to a very specific application setting.

Let's look at a straightforward way to benchmark an algorithm's runtime performance. This can be done by simply timing how long the algorithm takes to complete given various input sizes. As we mentioned earlier, this way of measuring runtime performance is dependent on the hardware that it is run on. Obviously, faster processors will give better results, however, the relative growth rates as we increase the input size will retain characteristics of the algorithm itself rather than the hardware it is run on. The absolute time values will differ between hardware (and software) platforms; however, their relative growth will still be bound by the time complexity of the algorithm.

Let's take a simple example of a nested loop. It should be fairly obvious that the time complexity of this algorithm is $O(n^2)$ since for each n iterations in the outer loop there are also n iterations in the interloop. For example, our simple nested for loop consists of a simple statement executed on the inner loop:

```
def nest(n):
for i in range(n):
     for j in range(n):
            i+j
```

The following code is a simple test function that runs the `nest` function with increasing values of `n`. With each iteration, we calculate the time this function takes to complete using the `timeit.timeit` function. The `timeit` function, in this example, takes three arguments, a string representation of the function to be timed, a `setup` function that imports the `nest` function, and an `int` parameter that indicates the number of times to execute the main statement.

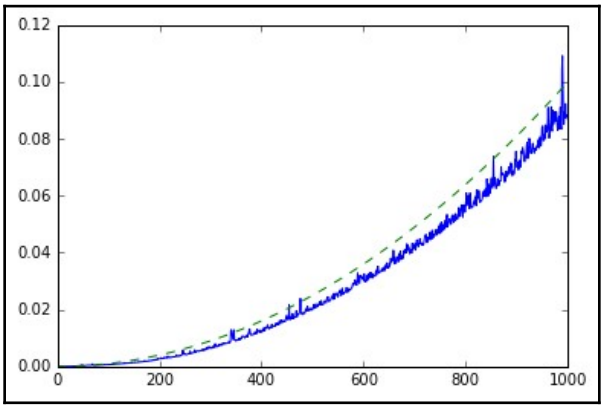
Since we are interested in the time the `nest` function takes to complete relative to the input size, `n`, it is sufficient, for our purposes, to call the `nest` function once on each iteration. The following function returns a list of the calculated runtimes for each value of `n`:

```
import timeit
def test2(n):
    ls=[]
    for n in range(n):
        t=timeit.timeit("nest(" + str(n) + ")", setup="from _main_ import
nest", number=1)
        ls.append(t)
    return ls
```

In the following code, we run the `test2` function and graph the results, together with the appropriately scaled n^2 function, for comparison, represented by the dashed line:

```
import matplotlib.pyplot as plt
n=1000
plt.plot(test2(n))
plt.plot([x*x/10000000 for x in range(n)])
```

This gives the following results:



As we can see, this gives us pretty much what we expect. It should be remembered that this represents both the performance of the algorithm itself as well as the behavior of underlying software and hardware platforms, as indicated by both the variability in the measured runtime and the relative magnitude of the runtime. Obviously, a faster processor will result in faster runtimes, and also performance will be affected by other running processes, memory constraints, clock speed, and so on.

Summary

In this chapter, we have looked at a general overview of algorithm design. Importantly, we studied a platform-independent way to measure an algorithm's performance. We looked at some different approaches to algorithmic problems. We looked at a way to recursively multiply large numbers and also a recursive approach for merge sort. We learned how to use backtracking for exhaustive search and generating strings. We also introduced the idea of benchmarking and a simple platform-dependent way to measure runtime.

In the following chapters, we will revisit many of these ideas with reference to specific data structures. In the next chapter, we will discuss linked lists and other pointer structures.