11

A SIMPLISTIC INTRODUCTION TO ALGORITHMIC COMPLEXITY

The most important thing to think about when designing and implementing a program is that it should produce results that can be relied upon. We want our bank balances to be calculated correctly. We want the fuel injectors in our automobiles to inject appropriate amounts of fuel. We would prefer that neither airplanes nor operating systems crash.

Sometimes performance is an important aspect of correctness. This is most obvious for programs that need to run in real time. A program that warns airplanes of potential obstructions needs to issue the warning before the obstructions are encountered. Performance can also affect the utility of many non-real-time programs. The number of transactions completed per minute is an important metric when evaluating the utility of database systems. Users care about the time required to start an application on their phone. Biologists care about how long their phylogenetic inference calculations take.

Writing efficient programs is not easy. The most straightforward solution is often not the most efficient. Computationally efficient algorithms often employ subtle tricks that can make them difficult to understand. Consequently, programmers often increase the **conceptual complexity** of a program in an effort to reduce its **computational complexity**. To do this in a sensible way, we need to understand how to go about estimating the computational complexity of a program. That is the topic of this chapter.

11.1 Thinking about Computational Complexity

How should one go about answering the question “How long will the following function take to run?”

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

We could run the program on some input and time it. But that wouldn't be particularly informative because the result would depend upon

- The speed of the computer on which it is run
- The efficiency of the Python implementation on that machine
- The value of the input

We get around the first two issues by using a more abstract measure of time. Instead of measuring time in microseconds, we measure time in terms of the number of basic steps executed by the program.

For simplicity, we will use a **random access machine** as our model of computation. In a random access machine, steps are executed sequentially, one at a time.[66] A **step** is an operation that takes a fixed amount of time, such as binding a variable to an object, making a comparison, executing an arithmetic operation, or accessing an object in memory.

Now that we have a more abstract way to think about the meaning of time, we turn to the question of dependence on the value of the input. We deal with that by moving away from expressing time complexity as a single number and instead relating it to the sizes of the inputs. This allows us to compare the efficiency of two algorithms by talking about how the running time of each grows with respect to the sizes of the inputs.

Of course, the actual running time of an algorithm can depend not only upon the sizes of the inputs but also upon their values. Consider, for example, the linear search algorithm implemented by

```
def linear_search(L, x):
   for e in L:
      if e == x:
         return True
   return False
```

Suppose that `L` is a list containing a million elements, and consider the call `linear_search(L, 3)`. If the first element in `L` is `3`, `linear_search` will return `True` almost immediately. On the other hand, if `3` is not in `L`, `linear_search` will have to examine all one million elements before returning `False`.

In general, there are three broad cases to think about:

- The best-case running time is the running time of the algorithm when the inputs are as favorable as possible. That is, the **best-case** running time is the minimum running time over all the possible inputs of a given size. For `linear_search`, the best-case running time is independent of the size of `L`.
- Similarly, the **worst-case** running time is the maximum running time over all the possible inputs of a given size. For `linear_search`, the worst-case running time is linear in the size of `L`.
- By analogy with the definitions of the best-case and worst-case running time, the **average-case** (also called **expected-case**) running time is the average running time over all possible inputs of a given size. Alternatively, if one has some *a priori* information about the distribution of input values (e.g., that `90%` of the time `x` is in `L`), one can take that into account.

People usually focus on the worst case. All engineers share a common article of faith, Murphy's Law: If something can go wrong, it will go wrong. The worst-case provides an **upper bound** on the running time. This is critical when there is a time constraint on how long a computation can take. It is not good enough to know that “most of the time” the air traffic control system warns of impending collisions before they occur.

Let's look at the worst-case running time of an iterative implementation of the factorial function:

```
def fact(n):
    """Assumes n is a positive int
       Returns n!"""
    answer = 1
    while n > 1:
       answer *=  n
       n -= 1
    return answer
```

The number of steps required to run this program is something like `2` (`1` for the initial assignment statement and `1` for the `return`) `+ 5n` (counting `1` step for the test in the `while`, `2` steps for the first assignment statement in the `while` loop, and `2` steps for the second assignment statement in the loop). So, for example, if `n` is `1000`, the function will execute roughly `5002` steps.

It should be immediately obvious that as `n` gets large, worrying about the difference between `5n` and `5n+2` is kind of silly. For this reason, we typically ignore additive constants when reasoning about running time. Multiplicative constants are more problematical. Should we care whether the computation takes `1000` steps or `5000` steps? Multiplicative factors can be important. Whether a search engine takes a half second or `2.5` seconds to service a query can be the difference between whether people use that search engine or go to a competitor.

On the other hand, when comparing two different algorithms, it is often the case that even multiplicative constants are irrelevant. Recall that in Chapter 3 we looked at two algorithms, exhaustive enumeration and bisection search, for finding an approximation to the square root of a floating-point number. Functions based on these algorithms are shown in Figure 11-1 and Figure 11-2.

```
def square_root_exhaustive(x, epsilon):
    """Assumes x and epsilon are positive floats & epsilon < 1
       Returns a y such that y*y is within epsilon of x"""
    step = epsilon**2
    ans = 0.0
    while abs(ans**2 - x) >= epsilon and ans*ans <= x:
        ans += step
    if ans*ans > x:
        raise ValueError
    return ans
```

Figure 11-1 Using exhaustive enumeration to approximate square root

```
def square_root_bi(x, epsilon):
    """Assumes x and epsilon are positive floats & epsilon < 1
       Returns a y such that y*y is within epsilon of x"""
    low = 0.0
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= epsilon:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans
```

Figure 11-2 Using bisection search to approximate square root

We saw that exhaustive enumeration was so slow as to be impractical for many combinations of values for `x` and `epsilon`. For example, evaluating `square_root_exhaustive(100, 0.0001)` requires roughly one billion iterations of the `while` loop. In contrast, evaluating `square_root_bi(100, 0.0001)` takes roughly 20 iterations of a slightly more complex `while` loop. When the difference in the number of iterations is this large, it doesn't really matter how many instructions are in the loop. That is, the multiplicative constants are irrelevant.

11.2 Asymptotic Notation

We use something called **asymptotic notation** to provide a formal way to talk about the relationship between the running time of an algorithm and the size of its inputs. The underlying motivation is that almost any algorithm is sufficiently efficient when run on small inputs. What we typically need to worry about is the efficiency of the algorithm when run on very large inputs. As a proxy for "very large," asymptotic notation describes the complexity of an algorithm as the size of its inputs approaches infinity.

Consider, for example, the code in Figure 11-3.

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0
    #Loop that takes constant time
    for i in range(1000):
        ans += 1
    print('Number of additions so far', ans)
    #Loop that takes time x
    for i in range(x):
        ans += 1
    print('Number of additions so far', ans)
    #Nested loops take time x**2
    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print('Number of additions so far', ans)
    return ans
```

Figure 11-3 Asymptotic complexity

If we assume that each line of code takes one unit of time to execute, the running time of this function can be described as `1000 + x + 2x`^{2}. The constant `1000` corresponds to the number of times the first loop is executed. The term `x` corresponds to the number of times the second loop is executed. Finally, the term `2x`^{2} corresponds to the time spent executing the two statements in the nested `for` loop. Consequently, the call `f(10)` will print

```
Number of additions so far 1000
Number of additions so far 1010
Number of additions so far 1210
```

and the call `f(1000)` will print

```
Number of additions so far 1000
Number of additions so far 2000
Number of additions so far 2002000
```

For small values of `x` the constant term dominates. If `x` is `10`, `over 80%` of the steps are accounted for by the first loop. On the other hand, if `x` is `1000`, each of the first two loops accounts for only about `0.05%` of the steps. When `x` is `1,000,000`, the first loop takes about `0.00000005%` of the total time and the second loop about `0.00005%`. A full `2,000,000,000,000` of the `2,000,001,001,000` steps are in the body of the inner `for` loop.

Clearly, we can get a meaningful notion of how long this code will take to run on very large inputs by considering only the inner loop, i.e., the quadratic component. Should we care about the fact that this loop takes $2x^2$ steps rather than x^2 steps? If your computer executes roughly 100 million steps per second, evaluating `f` will take about `5.5` hours. If we could reduce the complexity to x^2 steps, it would take about `2.25` hours. In either case, the moral is the same: we should probably look for a more efficient algorithm.

This kind of analysis leads us to use the following rules of thumb in describing the asymptotic complexity of an algorithm:

- If the running time is the sum of multiple terms, keep the one with the largest growth rate, and drop the others.
- If the remaining term is a product, drop any constants.

The most commonly used asymptotic notation is called "**Big O**" notation.[67] Big O notation is used to give an **upper bound** on the asymptotic growth (often called the **order of growth**) of a function. For example, the formula $f(x) \in O(x^2)$ means that the function `f` grows no faster than the quadratic polynomial x^2, in an asymptotic sense.

Many computer scientists will abuse Big O notation by making statements like, "the complexity of `f(x) is` $O(x^2)$." By this they mean that in the worst case `f` will take no more than $O(x^2)$ steps to run. The difference between a function being "in $O(x^2)$" and "being $O(x^2)$"

is subtle but important. Saying that $f(x) \in O(x^2)$ does not preclude the worst-case running time of `f` from being considerably less than $O(x^2)$. To avoid this kind of confusion we will use **Big Theta** (θ) when we are describing something that is both an upper and a **lower bound** on the asymptotic worst-case running time. This is called a **tight bound**.

Finger exercise: What is the asymptotic complexity of each of the following functions?

```
def g(L, e):
    """L a list of ints, e is an int"""
    for i in range(100):
        for e1 in L:
            if e1 == e:
                return True
    return False
def h(L, e):
    """L a list of ints, e is an int"""
    for i in range(e):
        for e1 in L:
            if e1 == e:
                return True
    return False
```

11.3 Some Important Complexity Classes

Some of the most common instances of Big O (and θ) are listed below. In each case, n is a measure of the size of the inputs to the function.

- $O(1)$ denotes **constant** running time.
- $O(\log n)$ denotes **logarithmic** running time.
- $O(n)$ denotes **linear** running time.
- $O(n \log n)$ denotes **log-linear** running time**.**
- $O(n^k)$ denotes **polynomial** running time. Notice that k is a constant.

- $O(c^n)$ denotes **exponential** running time. Here a constant is being raised to a power based on the size of the input.

11.3.1 Constant Complexity

This indicates that the asymptotic complexity is independent of the size of the inputs. There are very few interesting programs in this class, but all programs have pieces (for example, finding out the length of a Python list or multiplying two floating-point numbers) that fit into this class. Constant running time does not imply that there are no loops or recursive calls in the code, but it does imply that the number of iterations or recursive calls is independent of the size of the inputs.

11.3.2 Logarithmic Complexity

Such functions have a complexity that grows as the log of at least one of the inputs. Binary search, for example, is logarithmic in the length of the list being searched. (We will look at binary search and analyze its complexity in Chapter 12.) By the way, we don't care about the base of the log, since the difference between using one base and another is merely a constant multiplicative factor. For example, $O(\log_2(x)) = O(\log_2(10)*\log_{10}(x))$. There are lots of interesting functions with logarithmic complexity. Consider

```
def int_to_str(i):
    """Assumes i is a nonnegative int
       Returns a decimal string representation of i"""
    digits = '0123456789'
    if i == 0:
       return '0'
    result = ''
    while i > 0:
       result = digits[i%10] + result
       i = i//10
    return result
```

Since there are no function or method calls in this code, we know that we only have to look at the loops to determine the complexity class. There is only one loop, so the only thing that we need to do is characterize the number of iterations. That boils down to the number of times we can use `//` (floor division) to divide `i` by `10` before getting

a result of `0`. So, the complexity of `int_to_str` is in `O(log(i))`. More precisely, it is order θ(log(i)), because log(i) is a tight bound.

What about the complexity of

```
def add_digits(n):
    """Assumes n is a nonnegative int
       Returns the sum of the digits in n"""
    string_rep = int_to_str(n)
    val = 0
    for c in string_rep:
        val += int(c)
    return val
```

The complexity of converting `n` to a string using `int_to_str` is order θ`(log(n))`, and `int_to_str` returns a string of length `log(n)`. The `for` loop will be executed order θ`(len(string_rep))` times, i.e., order θ`(log(n))` times. Putting it all together, and assuming that a character representing a digit can be converted to an integer in constant time, the program will run in time proportional to θ`(log(n))` + θ`(log(n))`, which makes it order θ`(log(n))`.

11.3.3 Linear Complexity

Many algorithms that deal with lists or other kinds of sequences are linear because they touch each element of the sequence a constant (greater than `0`) number of times.

Consider, for example,

```
def add_digits(s):
    """Assumes s is a string of digits
       Returns the sum of the digits in s"""
    val = 0
    for c in string_rep:
        val += int(c)
    return val
```

This function is linear in the length of `s`, i.e., θ`(len(s))`.

Of course, a program does not need to have a loop to have linear complexity. Consider

```
def factorial(x):
    """Assumes that x is a positive int
       Returns x!"""
    if x == 1:
```

```
        return 1
    else:
        return x*factorial(x-1)
```

There are no loops in this code, so in order to analyze the complexity we need to figure out how many recursive calls are made. The series of calls is simply

```
factorial(x), factorial(x-1), factorial(x-2), … ,
factorial(1)
```

The length of this series, and thus the complexity of the function, is order θ`(x)`.

Thus far in this chapter we have looked only at the time complexity of our code. This is fine for algorithms that use a constant amount of space, but this implementation of factorial does not have that property. As we discussed in Chapter 4, each recursive call of `factorial` causes a new stack frame to be allocated, and that frame continues to occupy memory until the call returns. At the maximum depth of recursion, this code will have allocated `x` stack frames, so the space complexity is in `O(x)`.

The impact of space complexity is harder to appreciate than the impact of time complexity. Whether a program takes one minute or two minutes to complete is quite visible to its user, but whether it uses one megabyte or two megabytes of memory is largely invisible to users. This is why people typically give more attention to time complexity than to space complexity. The exception occurs when a program needs more space than is available in the fast memory of the machine on which it is run.

11.3.4 Log-Linear Complexity

This is slightly more complicated than the complexity classes we have looked at thus far. It involves the product of two terms, each of which depends upon the size of the inputs. It is an important class, because many practical algorithms are log-linear. The most commonly used log-linear algorithm is probably merge sort, which is order θ`(n log(n))`, where `n` is the length of the list being sorted. We will look at that algorithm and analyze its complexity in Chapter 12.

11.3.5 Polynomial Complexity

The most commonly used polynomial algorithms are **quadratic**, i.e., their complexity grows as the square of the size of their input. Consider, for example, the function in Figure 11-4, which implements a subset test.

```
def is_subset(L1, L2):
    """Assumes L1 and L2 are lists.
       Returns True if each element in L1 is also in L2
       and False otherwise."""
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

Figure 11-4 Implementation of subset test

Each time the inner loop is reached it is executed order θ(`len(L2)`) times. The function `is_subset` will execute the outer loop order θ(`len(L1)`) times, so the inner loop will be reached order θ(`len(L1)*len(L2)`).

Now consider the function `intersect` in Figure 11-5. The running time for the part of the code building the list that might contain duplicates is clearly order θ(`len(L1)*len(L2)`). At first glance, it appears that the part of the code that builds the duplicate-free list is linear in the length of `tmp`, but it is not.

```
def intersect(L1, L2):
    """Assumes: L1 and L2 are lists
       Returns a list without duplicates that is the intersection of
       L1 and L2"""
    #Build a list containing common elements
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
                break
    #Build a list without duplicates
    result = []
    for e in tmp:
        if e not in result:
            result.append(e)
    return result
```

Figure 11-5 Implementation of list intersection

Evaluating the expression `e not in result` potentially involves looking at each element in `result`, and is therefore θ(`len(result)`); consequently the second part of the implementation is order θ(`len(tmp)*len(result)`). However, since the lengths of `result` and `tmp` are bounded by the length of the smaller of `L1` and `L2`, and since we ignore additive terms, the complexity of `intersect` is θ(`len(L1)*len(L2)`).

11.3.6 Exponential Complexity

As we will see later in this book, many important problems are inherently exponential, i.e., solving them completely can require time that is exponential in the size of the input. This is unfortunate, since it rarely pays to write a program that has a reasonably high probability of taking exponential time to run. Consider, for example, the code in Figure 11-6.

```
def get_binary_rep(n, num_digits):
    """Assumes n and numDigits are non-negative ints
       Returns a str of length numDigits that is a binary
       representation of n"""
    result = ''
    while n > 0:
        result = str(n%2) + result
        n = n//2
    if len(result) > num_digits:
        raise ValueError('not enough digits')
    for i in range(num_digits - len(result)):
        result = '0' + result
    return result

def gen_powerset(L):
    """Assumes L is a list
       Returns a list of lists that contains all possible
       combinations of the elements of L. E.g., if
       L is [1, 2] it will return a list with elements
       [], [1], [2], and [1,2]."""
    powerset = []
    for i in range(0, 2**len(L)):
        bin_str = get_binary_rep(i, len(L))
        subset = []
        for j in range(len(L)):
            if bin_str[j] == '1':
                subset.append(L[j])
        powerset.append(subset)
    return powerset
```

Figure 11-6 Generating the power set

The function `gen_powerset(L)` returns a list of lists that contains all possible combinations of the elements of `L`. For example, if `L` is `['x', 'y']`, the powerset of `L` will be a list containing the lists `[]`, `['x']`, `['y']`, and `['x', 'y']`.

The algorithm is a bit subtle. Consider a list of `n` elements. We can represent any combination of elements by a string of `n` `0's` and `1's`, where a `1` represents the presence of an element and a `0` its absence. The combination containing no items is represented by a string of all `0's`, the combination containing all of the items is represented by a string of all `1's`, the combination containing only the first and last elements is represented by `100…001`, etc.

Generating all sublists of a list `L` of length `n` can be done as follows:

- Generate all `n`-bit binary numbers. These are the numbers from `0` to 2^n - 1.
- For each of these 2^n binary numbers, `b`, generate a list by selecting those elements of `L` that have an index corresponding to a `1` in `b`. For example, if `L` is `['x', 'y', 'z']` and `b` is `101`, generate the list `['x', 'z']`.

Try running `gen_powerset` on a list containing the first 10 letters of the alphabet. It will finish quite quickly and produce a list with 1024 elements. Next, try running `gen_powerset` on the first 20 letters of the alphabet. It will take more than a bit of time to run, and will return a list with about a million elements. If you try running `gen_powerset` on all 26 letters, you will probably get tired of waiting for it to complete, unless your computer runs out of memory trying to build a list with tens of millions of elements. Don't even think about trying to run `gen_powerset` on a list containing all uppercase and lowercase letters. Step 1 of the algorithm generates order $\theta(2^{len(L)})$ binary numbers, so the algorithm is exponential in len(L).

Does this mean that we cannot use computation to tackle exponentially hard problems? Absolutely not. It means that we have to find algorithms that provide approximate solutions to these problems or that find exact solutions on some instances of the problem. But that is a subject for later chapters.

11.3.7 Comparisons of Complexity Classes

The plots in this section are intended to convey an impression of the implications of an algorithm being in one or another of these complexity classes.

The plot on the left in Figure 11-7 compares the growth of a constant-time algorithm to that of a logarithmic algorithm. Note that the size of the input has to reach about 30,000 for the two of them to cross, even for the very small constant of 15. When the size of the input is 100,000, the time required by a logarithmic algorithm is still quite small. The moral is that logarithmic algorithms are almost as good as constant-time ones.

The plot on the right of Figure 11-7 illustrates the dramatic difference between logarithmic algorithms and linear algorithms.

While we needed to look at large inputs to appreciate the difference between constant-time and logarithmic-time algorithms, the difference between logarithmic-time and linear-time algorithms is apparent even on small inputs. The dramatic difference in the relative performance of logarithmic and linear algorithms does not mean that linear algorithms are bad. In fact, much of the time a linear algorithm is acceptably efficient.

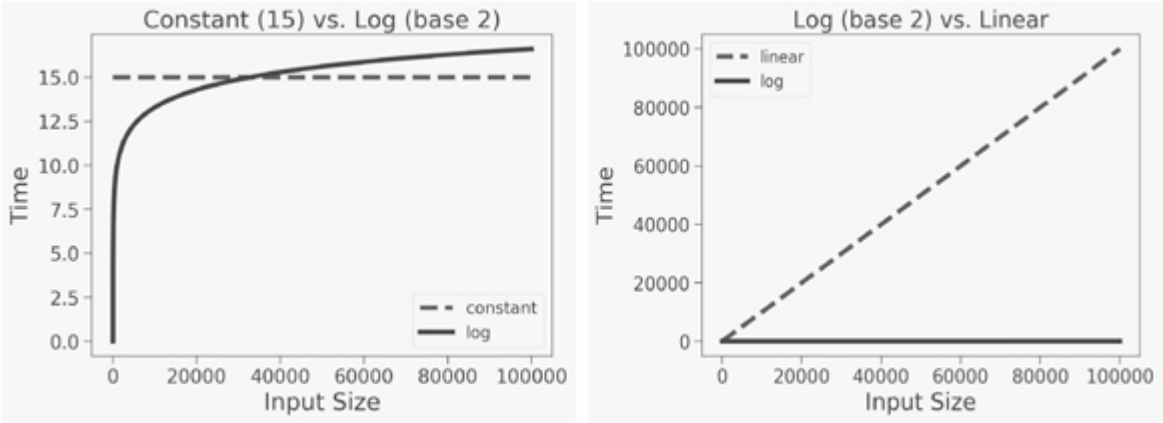

Figure 11-7 Constant, logarithmic, and linear growth

The plot on the left in Figure 11-8 shows that there is a significant difference between `O(n)` and `O(n log(n))`. Given how slowly `log(n)` grows, this may seem surprising, but keep in mind that it is a multiplicative factor. Also keep in mind that in many practical situations, `O(n log(n))` is fast enough to be useful. On the other hand, as the plot on the right in Figure 11-8 suggests, there are many situations in which a quadratic rate of growth is prohibitive.

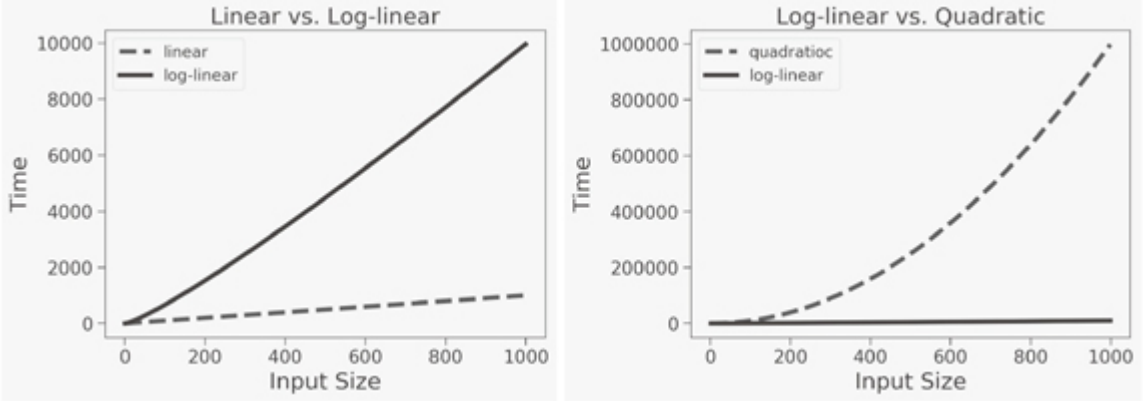

Figure 11-8 Linear, log-linear, and quadratic growth

The plots in Figure 11-9 are about exponential complexity. In the plot on the left of Figure 11-9, the numbers to the left of the y-axis run from `0` to `6`. However, the notation `1e29` on the top left means that each tick on the y-axis should be multiplied by 10^{29}. So, the plotted y-values range from `0` to roughly $6.6*10^{29}$. The curve for quadratic growth is almost invisible in the plot on the left in Figure 11-9. That's because an exponential function grows so quickly that relative to the `y` value of the highest point (which determines the scale of the y-axis), the `y` values of earlier points on the exponential curve (and all points on the quadratic curve) are almost indistinguishable from `0`.

The plot on the right in Figure 11-9 addresses this issue by using a logarithmic scale on the y-axis. One can readily see that exponential algorithms are impractical for all but the smallest of inputs.

Notice that when plotted on a logarithmic scale, an exponential curve appears as a straight line. We will have more to say about this in later chapters.

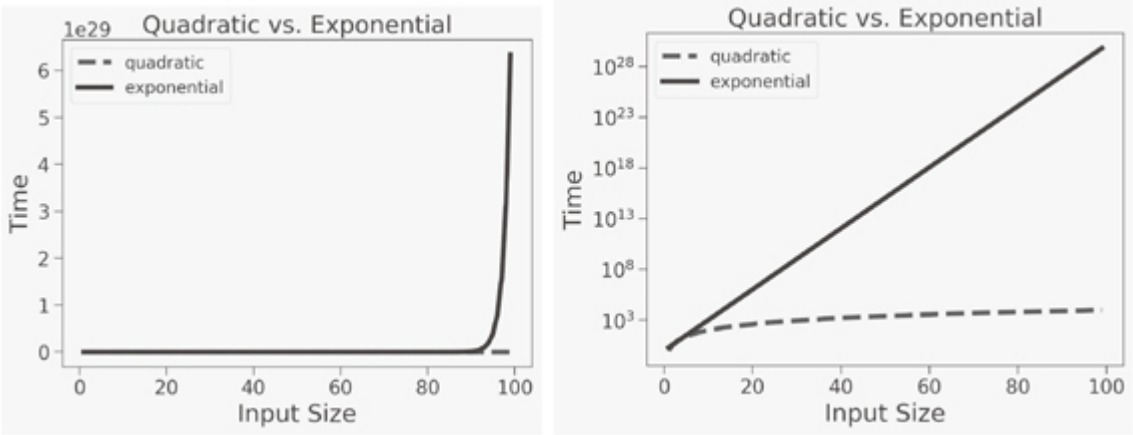

Figure 11-9 Quadratic and exponential growth

11.4 Terms Introduced in Chapter

conceptual complexity
computational complexity
random access machine
computational step
best-case complexity
worst-case complexity
average-case complexity
expected-case complexity
upper bound
asymptotic notation
Big O notation
order of growth
Big Theta notation
lower bound
tight bound

constant time

logarithmic time

linear time

log-linear time

polynomial time

quadratic time

exponential time

---

66 A more accurate model for today's computers might be a parallel random access machine. However, that adds considerable complexity to the algorithmic analysis, and often doesn't make an important qualitative difference in the answer.

67 The phrase “Big O” was introduced in this context by the computer scientist Donald Knuth in the 1970s. He chose the Greek letter Omicron because number theorists had used that letter since the late nineteenth century to denote a related concept.