*6.7 TIME COMPLEXITY

The algorithms that we are now designing have the potential to be applied to very large texts or even entire corpora. So this is an opportune time to revisit the time complexity of algorithms to develop a better understanding of what you might expect in these cases.

Recall that we analyze the time complexity of algorithms by counting the number of elementary steps that they require, in relation to the size of the input. An elementary step is one that takes the same amount of time every time we execute it, no matter what the input is. In Section 1.4, we gave the following examples of elementary steps:

- arithmetic operations,
- assignments of values to variables,
- testing a condition involving numbers or a character, and
- examining a character in a string or a number in a list.

Reflection 1 *Should a comparison between two strings, like* `word1 < word2`*, also count as an elementary step?*

To determine whether `word1 < word2`, the first characters must be compared, and then, if the first characters are equal, the second characters must be compared, etc. until either two characters are not the same or we reach the end of one of the strings. The total number of individual character comparisons depends on the values assigned to those variable names. Therefore, a string comparison may require *many* elementary steps, not just one. As we will discuss in the rest of this section, the time complexity of string operations must be considered carefully.

Best case vs. worst case

Let's look at string comparison more carefully.

Reflection 2 *If* `word1 = 'python'` *and* `word2 = 'rattlesnake'`*, how many character comparisons are necessary to conclude that* `word1 < word2` *is true? What if* `word1 = 'rattlesnake'` *and* `word2 = 'rattlesnakes'`*?*

In the first case, only a comparison of the first characters is required to determine that the expression is true. However, in the second case, or if the two strings are the same, we must compare every character to yield an answer. Therefore, assuming one string is not the empty string, the minimum number of comparisons is 1 and the maximum number of comparisons is n, where n is the length of the shorter string. (Exercise 6.7.1 more explicitly illustrates how a string comparison works.)

Put another way, the *best-case* time complexity of a string comparison is constant, or $\mathcal{O}(1)$, because it does not depend on the input size, and the *worst-case* time complexity for a string comparison is linearly proportional to n, or $\mathcal{O}(n)$.

Reflection 3 *Do you think the best-case time complexity or the worst-case time complexity is more representative of the true time complexity in practice?*

We are typically much more interested in the worst-case time complexity of an algorithm because it describes a guarantee on how long an algorithm can take. It also tends to be more representative of what happens in practice. For example, intuitively, the average number of character comparisons in a string comparison is about half the length of the shorter string, or $n/2$. Since $n/2$ is linearly proportional to n, the average case is also linear-time, making it more similar to the worst-case than the best-case.

Reflection 4 *What if one or both of the strings are constants, e.g., evaluating whether* `word == 'buzzards'`*?*

Because one of the values is a string literal with eight characters, we know that no more than eight character comparisons are *ever* required to decide whether the strings are equal, and this is independent of the input to any algorithm containing the comparison. Therefore, the time complexity of this operation is constant.

The algorithms that we have written in this chapter are a bit more complicated, at least at first glance. For example, let's look at the `findCharacter` function from Section 6.4, reproduced below.

```
1 def findCharacter(text, targetCharacter):
2     """ (docstring omitted) """

3     targetIndex = -1                        # assume it won't be found
4     for index in range(len(text)):
5         if text[index] == targetCharacter:  # if found, then
6             targetIndex = index             #   remember where
7             break                           #   and exit the loop early
8     return targetIndex
```

The first and last statements in the function, on lines 3 and 8, are elementary steps because their times are independent of the value of the input, `text`. The rest of the work in the function is done by the `for` loop on lines 4–7.

Reflection 5 *Suppose that* `text` *contains n characters (i.e.,* `len(text)` *is n). In terms of n, how many times does the* `for` *loop iterate?*

The loop iterates n times, once for each character in `text`. In each of these iterations, an integer is implicitly assigned to the variable `index`, which we count as one elementary step per iteration. Then the comparison in the `if` statement on line 5 is executed. Since both `text[index]` and `targetCharacter` are single characters, this is one more elementary step per iteration. The two elementary steps on lines 6–7, which are only executed if the condition is true, also end the loop. So each iteration of the `for` loop will execute between two and four elementary steps, but the latter can only happen once.

Reflection 6 *Overall how many elementary steps does this function execute in the best and worst cases?*

In the best case, the `if` condition in the `for` loop will be true right away, causing the loop to end after only one iteration. Overall then, the function will execute one elementary step in line 3, plus four elementary steps in its one iteration of the loop, plus one elementary step in line 8, for a total of six elementary steps. This means that the function is a constant-time, or $\mathcal{O}(1)$, algorithm in the best case.

In the worst case, the `if` condition will be false during the first $n-1$ iterations, allowing the loop to reach its last possible iteration. The `for` loop will execute two elementary steps in each of these $n-1$ iterations, for a total of $2(n-1)$. The final, n-th iteration will execute between two and four elementary steps, depending on whether the `if` condition is true.

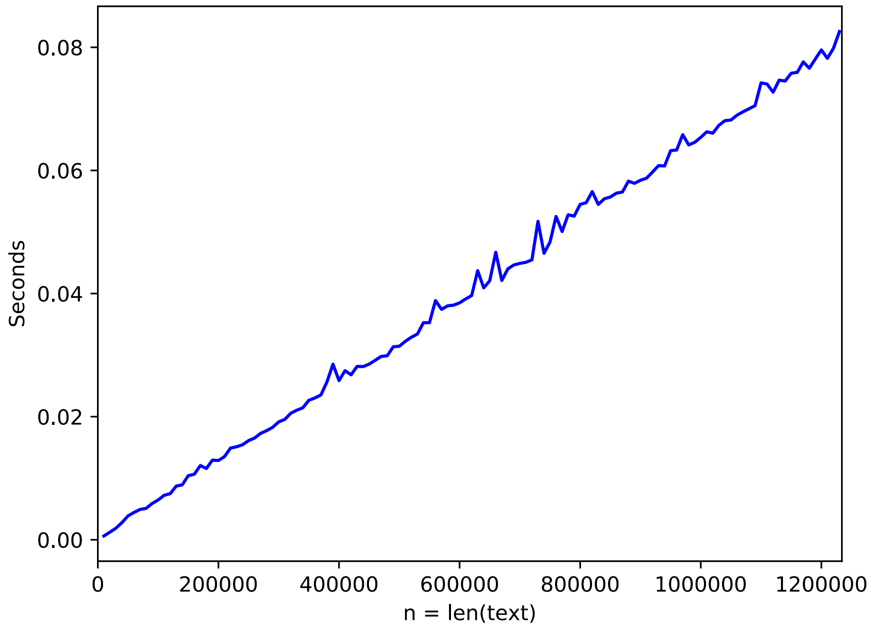

Figure 1 An empirical analysis of the time complexity of the `findCharacter` function.

Therefore, overall the function will execute $2 + 2(n - 1) + 4 = 2n + 4$ elementary steps in the worst case, meaning that it is a linear-time, or $\mathcal{O}(n)$, algorithm.

For this reason, the algorithmic technique used by `findCharacter` (and the `find` function from the same section) is known as a ***linear search*** (or *sequential search*). In Chapter 10, we will see an alternative search algorithm that is much faster, but it can only be used in cases where the data is maintained in sorted order.

This analysis makes perfect sense if you think about it. In the best case, the `findCharacter` function will find the character it seeks in the first character of `text` and finish right away. On the other hand, in the worst case, it will search through the entire `text` and either not find the character it seeks or find it in the very last character in `text`.

To see if this analysis holds up in practice, we timed `findCharacter` on increasingly long portions of the text of Herman Melville's *Moby Dick*. To simulate the worst-case behavior, we searched for a control character that is almost definitely not found in any ordinary text file. The results of this experiment are shown in Figure 1. The x-axis of the plot represents the length of the text segments that we used, from the first 10,000 characters up to the entire book, in 10,000-character increments. You can see that the curve is pretty close to a straight line, indicating a linear time complexity.

Asymptotic time complexity

The jagged nature of the line in Figure 1 is due to the variable workload on the computer while this program ran. Every time we run a program like this, the time it takes will be a little different. And if we ran it on different computers, it would be even more different.

But the linear *rate* at which the time grows with input size will be the same regardless. Intuitively, this is one reason why we are only concerned with general classifications of algorithms like constant-time and linear-time, rather than the exact number of elementary steps they require. A related reason is that different elementary steps take different amounts of time on different computers and in different programming languages. So the distinction between something like $2n + 4$ and $5n + 2$ elementary steps is really meaningless. It could be the case that the algorithm with $5n + 2$ elementary steps is actually a little faster in practice because the elementary steps that it uses are faster than the ones the other algorithm uses. So all that really matters is that both time complexities are linear functions of n.

As we discussed briefly back in Section 1.4, understanding the ***scalability*** of algorithms is the most important reason we are only interested in general classifications of time complexity. In other words, we are really only concerned with how well an algorithm will cope when inputs scale up to huge sizes. Intuitively, this is because virtually all algorithms are going to be very fast when the input sizes are small. Indeed, we saw in Figure 1 that the running time of `findCharacter` on the entire text of *Moby Dick*, about 1.2 million characters, was less than one-tenth of a second. However, when input sizes get really large, differences in time complexity can become quite significant.

We formally call this idea ***asymptotic time complexity***. *Asymptotic* refers to our interest in arbitrarily large input sizes. An asymptote, a line that an infinite curve gets arbitrarily close to, but never actually touches, should be familiar if you have taken some calculus.

In asymptotic time complexity, we simplify $2n + 4$ to $\mathcal{O}(n)$ because the other terms in the expression have virtually no impact on the growth rate as the input size grows very large. To make this concrete, let's compare the growth rates of $2n + 4$ and n.

n	$2n + 4$	growth factor
10	24	—
100	204	8.5000
1,000	2,004	9.8235
10,000	20,004	9.9820
100,000	200,004	9.9982
1,000,000	2,000,004	9.9998
10,000,000	20,000,004	9.9999

The first column of the table contains exponentially increasing values of n. In each row, the value of n is growing by a factor of ten. Compare this to the third column, which shows the rate at which $2n+4$ is growing. In the second row, $2n+4$ grew by a factor of $204/24 = 8.5$. In the third row, it grew by a factor of $2{,}004/204 = 9.8235$. As n grows larger and larger, this ratio gets closer and closer to ten. In other words, for very large values of n, n and $2n + 4$ grow at essentially identical rates! So the constants 2 and 4 in the expression $2n + 4$ have virtually no impact at all on the rate at which the expression grows when n gets large. By saying that $2n + 4$ is asymptotically $\mathcal{O}(n)$, we are saying that $2n + 4$ is really the same thing as n, as n approaches infinity.

Understanding this, dealing with asymptotic time complexity actually makes analyzing algorithms quite a bit simpler! Consider the `splitIntoWords` function from Section 6.1, reproduced below.

```
 1 def splitIntoWords(text):
 2     """ (docstring omitted) """

 3     wordList = []
 4     prevCharacter = ' '
 5     word = ''
 6     for character in text:
 7         if character not in string.whitespace:
 8             word = word + character
 9         elif prevCharacter not in string.whitespace:
10             wordList.append(word)
11             word = ''
12         prevCharacter = character
13     if word != '':
14         wordList.append(word)
15     return wordList
```

This function is dominated by a `for` loop that iterates n times, where n=`len(text)`, the size of the input. Therefore, as long as each iteration of the loop contains a constant number of elementary steps and all of the statements outside the loop don't comprise more than $\mathcal{O}(n)$ elementary steps, this is a linear-time algorithm. In other words, we don't need to worry about the exact number of elementary steps in the body of the loop, just that the number is constant. This is indeed the case because each of lines 7–12 can be considered to be a constant number of elementary steps.[3] Outside the loop, the same can be said for lines 3–5 and 13–15. Therefore the worst-case asymptotic time complexity of this algorithm is $\mathcal{O}(n)$! We leave it as an exercise for you to support each of these assertions in more detail.

As another example, consider this partial `dotplot` function from Section 6.6.

```
 1 def dotplot(text1, text2):
 2     """ (docstring omitted) """

 3     text1 = text1.lower()
 4     text2 = text2.lower()
 5     x = []
 6     y = []
 7     for index1 in range(len(text1)):
 8         for index2 in range(len(text2)):
 9             if text1[index1] == text2[index2]:
10                 x.append(index1)
11                 y.append(index2)
12     # plot x and y (omitted)
```

To simplify things a bit, assume that `text1` and `text2` are the same length, so n represents the length of both strings. Let's start at the top with the two calls to the `lower` method in lines 3–4.

[3]Line 8 would appear to be a linear-time operation because it involves copying the characters in the previous value of `word`. However, a clever trick is employed in the Python interpreter that makes each concatenation in a long sequence of concatenations a constant-time operation on average. A more in-depth discussion of this, in the context of lists, can be found in Section 7.5.

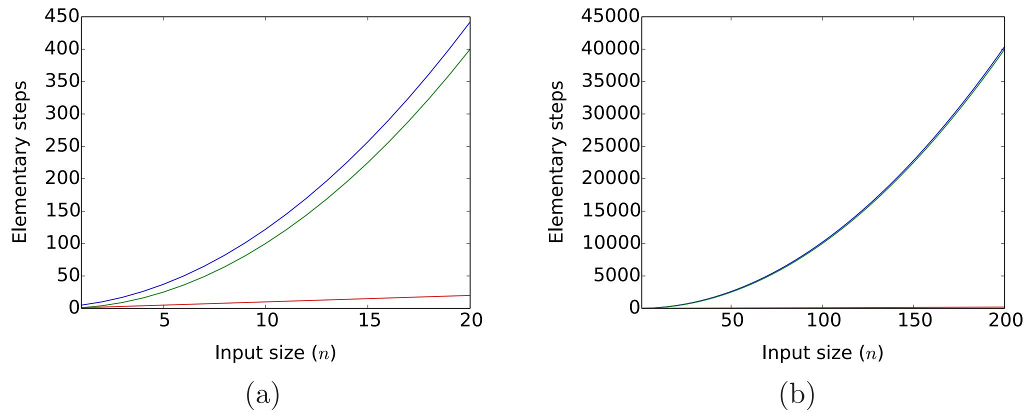

Figure 2 Two views comparing $n^2 + 2n + 2$ (blue), n^2 (green), and n (red).

Reflection 7 *Are lines 3 and 4 elementary steps? If not, about how many elementary steps do they require?*

Since the `lower` method needs to consider every character in the string, each of these lines must require about n elementary steps. The two lines following are each one elementary step. The main part of the function is a `for` loop in lines 8–11 that is nested inside another `for` loop. The inner `for` loop iterates n times and each of the statements in the body of the loop is an elementary step, so the inner `for` loop contains at most $3n$ elementary steps. In each of the n iterations of the outer `for` loop on line 7, the inner `for` loop is executed one time. Therefore, the total number of elementary steps in the nested loops is $n \cdot 3n = 3n^2$. Altogether then, the function contains about $2n + 2 + 3n^2$ elementary steps.

Since n^2 grows faster than both $2n$ and 2, we can ignore both of those terms, and say that this algorithm's asymptotic time complexity is $\mathcal{O}(n^2)$. The intuition behind this asymptotic time complexity is illustrated in Figure 2. When we view $n^2 + 2n + 2$ and n^2 together for very small values of n in Figure 2(a), it appears as if $n^2 + 2n + 2$ is diverging from n^2, but when we "zoom out," looking at values of n that are just a little larger, as in Figure 2(b), we see that the functions are almost indistinguishable. In both graphs, we can also see that both $n^2 + 2n + 2$ and n^2 are significantly different than a linear time complexity, which is almost indistinguishable from the x-axis.

Algorithms like `dotplot` that have $\mathcal{O}(n^2)$ asymptotic time complexity are said to be ***quadratic-time algorithms***.

Another way to visualize the difference between a linear-time algorithm and a quadratic-time algorithm is shown in Figure 3. Suppose each square represents one elementary step. On the left is a representation of the work required in a linear-time algorithm. If the size of the input to the linear-time algorithm increases from $n - 1$ to n ($n = 7$ in the pictures), then the algorithm must execute one additional elementary step (in gray). On the right is a representation of the work involved in a quadratic-time algorithm. If the size of the input to the quadratic-time algorithm increases from $n - 1$ to n, then the algorithm gains $2n - 1$ additional steps. So we can see that the amount of work in a quadratic-time algorithm grows much more quickly than the work required by a linear-time algorithm!

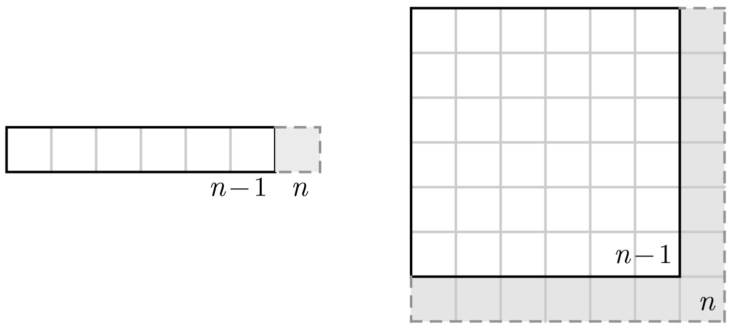

Figure 3 Linear vs. quadratic growth.

Exercises

6.7.1. The following function more explicitly illustrates how a string comparison works "behind the scenes." The `comparisons` variable counts how many individual character comparisons are made.

```
def compare(word1, word2):
    index = 0
    comparisons = 3
    while (index < len(word1)) and (index < len(word2)) \
                               and (word1[index] == word2[index]):
        index = index + 1
        comparisons = comparisons + 3

    if index == len(word1) and index == len(word2):   # case 1: ==
        result = 'equal'
        comparisons = comparisons + 2
    elif index == len(word1) and index < len(word2):  # case 2: <
        result = 'less'
        comparisons = comparisons + 2
    elif index == len(word2) and index < len(word1):  # case 3: >
        result = 'greater'
        comparisons = comparisons + 2
    elif word1[index] < word2[index]:                 # case 4: <
        result = 'less'
        comparisons = comparisons + 1
    else:                                             # case 5: >
        result = 'greater'
        comparisons = comparisons + 1

    return result, comparisons

def main():
    word1 = 'canny'
    word2 = 'candidate'
    result, comparisons = compare(word1, word2)
    if result == 'less':
        print(word1 + ' comes before ' + word2 + '.')
    elif result == 'greater':
        print(word1 + ' comes after ' + word2 + '.')
    else:
        print(word1 + ' and ' + word2 + ' are equal.')

    print(str(comparisons) + ' comparisons were made.')

main()
```

Study and experiment with this function and answer the following questions.

(a) Explain why each of the five cases gives the result that it does.

(b) How many comparisons happen in the `compare` function when

   i. `'canny'` is compared to `'candidate'`?
   ii. `'canny'` is compared to `'danny'`?
   iii. `'canny'` is compared to `'canny'`?
   iv. `'can'` is compared to `'canada'`?
   v. `'canoeing'` is compared to `'canoe'`?

(c) Suppose `word1` and `word2` are the same n-character string. How many comparisons happen when `word1` is compared to `word2`?

(d) Suppose `word1` (with m characters) is a prefix of `word2` (with n characters). How many comparisons happen when `word1` is compared to `word2`?

(e) The value of `comparisons` actually over-counts the number of comparisons in some cases. When does this happen?

6.7.2. For each of the following code snippets, think carefully about how it must work, and then indicate whether it represents a constant-time algorithm, a linear-time algorithm, or a quadratic-time algorithm. The variable `name` refers to a string object with length n. Justify each of your answers.

(a)* `name = name.upper()`

(b)* `name = name.find('x')`

(c)* `name = 'accident'.find('x')`

(d) `newName = name.replace('a', 'e')`

(e) `newName = name + 'son'`

(f) `newName = 'jack' + 'son'`

(g) `index = ord('H') - ord('A') + 1`

(h)*
```
for character in name:
    print(character)
```

(i)
```
for character in 'hello':
    print(character)
```

(j)
```
if name == newName:
    print('yes')
```

(k)
```
if name == 'hello':
    print('yes')
```

(l)
```
if 'x' in name:
    print('yes')
```

(m)
```
for character in name:
    x = name.find(character)
```

6.7.3. The following function performs rough worst-case timing experiments of a test function on increasingly long segments of a dummy text. It uses the `time.time()` function (in the `time` module), which returns the current time in seconds elapsed since January 1, 1970.

```
import time

def timing(maxLength, stepLength):
    """Plot timing experiments on a text analysis function.

    Parameters:
        maxLength:  the maximum length of the test text
        stepLength: steps between test text lengths

    Return value: None
    """

    text = 'a' * maxLength   # a dummy text
    times = []

    for length in range(stepLength, maxLength, stepLength):
        testText = text[:length]
        begin = time.time()

        # call a function to time here with argument(s) testText

        end = time.time()
        times.append(end - begin)   # append elapsed time

    pyplot.plot(range(stepLength, maxLength, stepLength), times)
    pyplot.xlabel('n = len(text)')
    pyplot.ylabel('Seconds')
    pyplot.show()
```

The plot produced by this function is an empirically derived time complexity plot similar to that in Figure 1. Use this function to perform timing experiments with each of the following functions below. For (a)–(c), use `timing(500000, 10000)`. For (d), use `timing(5000, 100)`. Discuss each result, including what it appears to indicate about the asymptotic time complexity of the function.

(a)* the `fletcherChecksum` function on page 249

(b) the `wordFrequency` function on page 268

(c) the `copy` function on page 225

(d) the truncated `dotplot` function on page O6.7-5 (use `testText` for both arguments)

6.7.4. Decide whether each of the following functions from this chapter is asymptotically a constant-time, linear-time, or quadratic-time algorithm. Explain each of your answers.

(a)* the `copy` function on page 225 (see the footnote on page O6.7-5)

(b) the `removePunctuation` function on page 227 (see the footnote on page O6.7-5)

(c) the `normalize` function on page 227

(d)* the `wordTokens` function on page 231

(e) the `wordCount` function on page 232

(f) the `fletcherChecksum` function on page 249

(g) the `wordFrequency` function on page 268

(h) the `sliceFrequencies` function on page 270

6.7.5. What is the asymptotic time complexity of an algorithm that requires each of the following numbers of elementary steps? Assume that n is the length of the input in each case.

(a) $7n - 4$

(b) 6

(c) $3n^2 + 2n + 6$

(d) $4n^3 + 5n + 2^n$

(e) $n \log_2 n + 2n$

6.7.6. Suppose that two algorithms for the same problem require $12n$ and n^2 elementary steps. On a computer capable of executing 1 billion steps per second, how long will each algorithm take (in seconds) on inputs of size $n = 10$, 10^2, 10^4, 10^6, and 10^9? Is the algorithm that requires n^2 steps ever faster than the algorithm that requires $12n$ steps?

Selected Exercise Solutions

6.7.2 (a) linear time

(b) linear time

(c) constant time

(h) linear time

6.7.3 (a) Calling `timing(500000, 10000)` with `fletcherChecksum(testText)` between the calls to `time.time()` yields a plot that resembles a straight diagonal line, suggesting a linear time complexity.

6.7.4 (a) linear-time

(d) linear-time